Shopping Models: A Flexible Architecture for Information Commerce *

Steven P. Ketchpel Hector Garcia-Molina Andreas Paepcke
Stanford University
Computer Science Department
Stanford, CA 94305
{ketchpel, hector, paepcke}@cs.stanford.edu

Abstract

In a digital library, there are many different interaction models between customers and information providers or merchants. Subscriptions, sessions, pay-per-view, shareware, and pre-paid vouchers are different models that each have different properties. A single merchant may use several of them. Yet if a merchant wants to support multiple models, there is a substantial amount of work to implement each one. In this paper, we formalize the *shopping models* which represent these different modes of consumer to merchant interaction. In addition to developing the overall architecture, we define the application program interfaces (API) to interact with the models. We show how a small number of primitives can be used to construct a wide range of shopping models that a digital library can support, and provide examples of the shopping models in operation, demonstrating their flexibility.

1 Introduction

With the exponential growth of information, the economics of libraries are changing. Library budgets are flat or decreasing, yet libraries are expected to provide access to the full range of materials. Practices such as university libraries sharing access to less common journals are one way to stretch available resources. Other ways that may be more effective in the digital environment are pay-per-view, where the library does not have to pay for materials until a user explicitly requests them, or cost-sharing, where the library patron pays some of the cost of the requested information. In other cases, the variability of a pay-per-view model is unacceptable. A fixed cost subscription rate may be more cost-effective or ease the uncertainty in budget demands.

Digital libraries may not always have a public, corporate, or academic institution intervening between a patron and the information providers. In some cases, the publishers may be selling information directly to the end consumer. In other cases, the intermediary may be an add-on (outsourced) service paid for by the user or an employer.

Another significant change in the information landscape is the increased use of "push" models to distribute information and infomercials in both narrowly targeted and broadly disseminated media. The mixture of advertising with "content" can mitigate the cost of the content, and invites a whole different set of distribution models.

Given this range of emerging opportunities, it is clear that different transactions in the electronic world are done according to very different guidelines. The choice of the appropriate methods is affected by issues of trust between the customer and merchant, as well as convenience. Currently, implementing interaction models for electronic document commerce is a time-consuming, resource-intensive process. Internet commerce servers (such as those from Netscape, Microsoft and Open Market) provide one model with minimal flexibility to implement other models. Even if these servers wished to provide multiple modes of interaction, it is not at

*This material is based upon work supported by the National Science Foundation under Cooperative Agreement IRI-9411306. Funding for this cooperative agreement is also provided by DARPA, NASA, and the industrial partners of the Stanford Digital Libraries Project. The first author was partially supported by a National Defense Science and Engineering Graduate Fellowship.

all clear how they would do it, other than by providing separate, incompatible interfaces. Furthermore, if a customer wishes to interact with servers that offer different models (e.g., subscriptions, pay-per-view), his code must handle all the protocols, an approach that is not only extremely expensive but also does not scale.

The goal of this paper is to define an application program interface (API) for merchants and customers for describing these different models of interaction. By breaking down these customer/merchant interactions into a small set of primitives, we show that a merchant or a customer only need to provide some basic functionality for ordering goods, payment, and delivery, *independent of the type of shopping interaction.* The programming logic for the shopping interaction can be encapsulated into a *shopping model* entity that can be shared among customers and merchants that have a need for that model.

Our approach could be used by a single merchant who wishes to implement multiple models of interaction in a clean way. However, it could also be adopted as a standard interaction framework between merchants and clients, making it possible for a client to interact with many merchants under many different interaction models in a uniform way.

Due to space limitations, we will not be able to offer a complete presentation of all of the details of the system. Instead, we settle for giving the reader an overview of the main concepts, providing supporting details where space permits. One important issue we do not cover in this short paper is security, although we briefly discuss it in Section 7.

2 Shopping Model: The Concept

Consider five different examples:

1. *Subscription*: A customer orders 52 issues of *BusinessWeek*, and starts receiving them after paying the bill.

2. *Session*: A customer enrolls in the Knight Ridder Information System, entitling him to make queries throughout the month, receiving an itemized bill at the end of the period, which he pays in order to receive continued service.

3. *Pay-Per-View*: A customer hears about a special keynote address being broadcast in scrambled form on the internet MBONE. By signing up in advance, and sending her credit card number, she is provided a key which allows her to descramble the transmission.

4. *Shareware*: A customer subscribes to a "pay-what-you-will" newsletter. After a particularly enlightening issue, the reader decides to make a donation to the volunteer editor.

5. *Pre-paid voucher*: A research analyst is instructed to produce a report for a corporate client. Rather than allowing the expenses to be billed back to the requestor, the company stipulates a hard limit by paying \$3,000 to the information service, and giving the resulting receipt to the analyst as a pre-paid voucher which he will spend down.

Each of these examples shows a different model of interaction between a *customer* ordering information and a *merchant* who provides it. Although the interactions are very different, from the point of view of the participants there is some common functionality that must be supported in all cases. For example, the merchant needs to "check a purchase order for consistency," or he may need to "initiate the delivery of goods." Depending on the shopping interaction, these merchant operations may have different parameters (e.g., the details of the order or the good to deliver will vary). Furthermore, the operations may occur in different relative orders, or different numbers of times. For instance, for a promotion, goods may be delivered before the customer places an order. For a subscription interaction, goods may be delivered multiple times. Yet, in spite of these differences, there are basic, common functions a merchant must provide under all types of shopping interactions. Similarly, there are common functions that the customer and other components (e.g., a bank involved in the interaction) must provide.

This insight is the key to the *shopping model* abstraction. That is, our goal is to separate the program logic and state into four components: that which deals with the merchant, that which deals with the customer, that which deals with other services (such as payment and delivery), and that which is specific to a particular shopping transaction we are handling. We refer to the code and state that coordinates a particular shopping

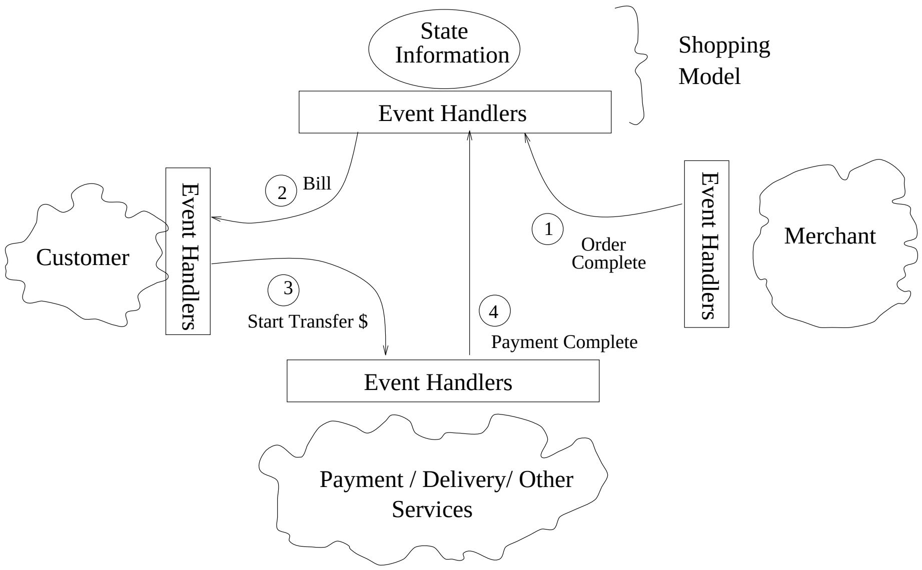

Figure 1: A high level view of a shopping model

transaction, as the *shopping model*. The shopping model tells the participants (e.g., merchant, customer) what to do next in the way of ordering, payment, and delivery. The customer and merchant mutually agree upon what shopping model to use, and the shopping model is typically run by either the merchant or a trusted third party.

We rely on *event-driven programming* for our design. All participants, including the shopping model, respond to incoming messages by taking local actions (such as authorizing an order or verifying a payment) and sending other messages to advance the state of the shopping transaction. Figure 1 illustrates the participants in a transaction, and how the logic of the particular shopping model is carried out. The "clouds" in the figure represent the *participants*: merchant, customer, and other services. Each participant has a set of *event handlers* that deal with requests for that participant. The shopping model at the top of the figure also provides event handlers to deal with actions that require shopping model-specific coordination. These handlers rely on state information that records the current situation, e.g., what steps have been performed so far in the interaction.

To illustrate how the event handlers work together in an interaction, Figure 1 also shows a fragment of a particular transaction. We start the example after the merchant has received a completed order. (We will discuss later how we arrived at this state.) After checking that the order is valid, the merchant's event handler contacts the shopping model (Message 1), informing the shopping model that one step has been completed. The shopping model determines what to do next, based on the behavior encoded in its event handlers. For this particular shopping model, let us say that the step that follows a completed order is the customer's payment. Therefore, the shopping model's event handler sends a message (Message 2) to the customer's event handler, requesting that payment be made in order to continue this transaction. This event handler decides if payment is appropriate from the customer's point of view (e.g., is the customer willing to pay the requested amount). If so, the customer handler contacts the handler for the specific payment mechanism he wishes to use (Message 3), asking it to transfer money from the user's account into the merchant's account. Notice how we have separated the client side of payment, from the low level details of the actual financial operation. After the handler for the payment mechanism completes payment, it notifies the shopping model (Message 4) that the step following payment may be started.

The event handlers for the customer, merchant, and other services can be thought of as *proxies* or *wrappers* to autonomous systems. For instance, when Message 3 in Figure 1 arrives, the handler may have to convert the request for payment into one or more messages to the underlying payment service, such as DigiCash or First Virtual. These messages must be in the format and must use the protocol specific to the service. When the service responds, e.g., saying the funds transfer has completed, the event handler converts the message into the common format we are proposing in this paper and sends out Message 4. This use of proxies insulates the shopping model and the other participants from the implementation details for a particular action. Hence, neither the shopping model nor the customer need to know how to make a payment using DigiCash[2] or First Virtual[7] or how to send goods enclosed in a Cryptolope[4] or DigiBox[6]. Similarly, the responses to service requests are at a high level of abstraction. For instance, rather than hearing that the public key server was unavailable, so the DigiCash coins could not be verified, the shopping model just learns that "Payment Failed".

Of course, the event handlers neeed not be proxies. A participant, say a particular merchant, could directly provide the event handlers we propose in this paper. In this case, a shopping transaction could interact directly with the merchant application, without a need to translate requests. Our ultimate goal in proposing our shopping model framework is that participants adopt it (or something similar) as a "standard" way of coordinating very general types of information and commerce transactions.

It is important to notice that shopping models can be generic. That is, an annual subscription to *BusinessWeek* could use the same shopping model code and data structures as an annual subscription to *TV Guide*. By the conventions of subscriptions, both publishers receive an order, wait for payment, then provide a repeated delivery. Of course, there has to be a difference in the actual operation of the subscriptions, but the differences can be contained exclusively in the set of merchant event handlers. For example, a merchant payment event handler makes sure the money is deposited in the *BusinessWeek* account rather than *TV Guide's*. Similarly, a merchant delivery event handler controls the content that is distributed to the customers. Generic shopping models make it possible to think of a "library" of shopping models, where either customers or merchants could search for useful ones, and then ask the other parties if they are willing to operate under a proposed model. Such shopping model negotiation will be discussed later.

For modularity we functionally divide the event handlers of each participant (including the shopping model) into three classes: those dealing with orders, those dealing with payments, and those dealing with delivery. We represent each class of handlers for a given participant as an object with methods representing the events that can be handled. For example, the `CustomerPayment` handler object is in charge of customer related payment issues; its methods include `ChoosePaymentMethod`, to select a preferred payment method from a list of choices, and `Pay`, called when the customer needs to pay. The merchant also has a payment handler object, `MerchantPayment`, but it deals with merchant related issues. Similarly, the shopping model's payment handler, `ShoppingPayment` has methods to decide what action must be followed when payment is requested or completed or not possible. Since we have four participants, identified by `Customer, Merchant, Services, Shopping`, and three types of handler objects, `Payment, Delivery, Order`, we have a total of 12 event handler objects.

In the Appendix we provide a complete definition for all these objects and their methods. The definitions are given in ISL, a CORBA-like interface specification language. The use of CORBA for defining the objects and their methods provides a level of platform and transport independence that removes many of the problems of distributed application development.

For a particular transaction between a specific customer and a specific merchant, a *Shopping Transaction Record (STR)* (shown at the top left of Figure 2) maintains the context. It is available with every method invocation, either passed directly as an argument, or via a context pointer of another argument. The STR contains:

- references to all of the shopping model, merchant, and customer event handlers
- a reference to the order (goods, price, terms and conditions, delivery mechanisms, payment mechanisms, etc.)
- lists of handlers that should be notified of status updates for payment or delivery
- any additional local state necessary (e.g., number of issues remaining)

Notice that different STR's may reference different handlers *for the same participant*. This makes it possible to, say, use different `MerchantOrder` event objects if the transaction is for a subscription as opposed to a pay-per-view interaction. If, however, the merchant has a single type of `MerchantOrder` object to handle all orders, then all STR instances will point to this same object.

The STR also contains a reference to the `Order` record. This record describes the goods that were requested by this transaction. It includes a description of the goods, the price, a list of acceptable payment

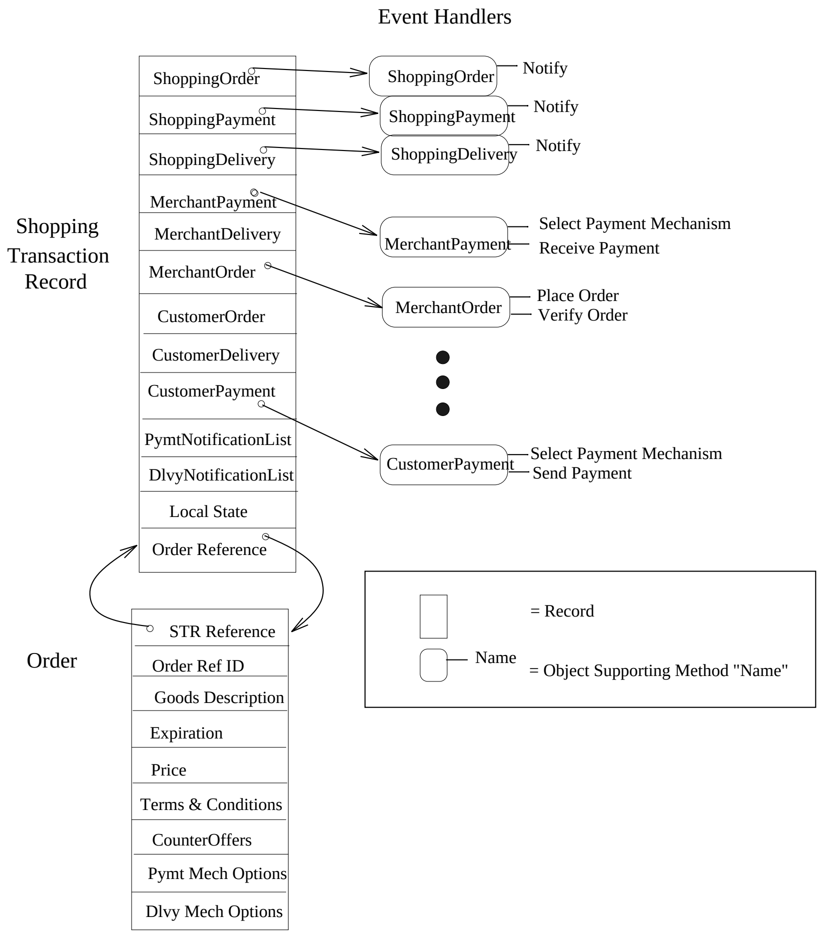

Figure 2: Relation of Main Records and Objects

services, and so on. Finally, the STR contains a reference to a local state record that holds additional information regarding this transaction. (For simplicity, in this paper we leave the structure of the local state unspecified.) The entire STR record constitutes the "state information" that was shown at the top of Figure 1. In the following section we will show how the various STR fields are initialized as a shopping transaction starts.

3 Shopping Models: Examples

Since the interplay of the various objects can be difficult to describe and understand in the general case, we will look at some specific examples, and show how the shopping model framework would be applied. The first is that of a pre-paid magazine subscription, and is considered in some detail. The subsequent examples (sessions and shareware) are abbreviated for the sake of space.

3.1 Subscription Model

The sequence of messages among the various roles of the merchant and consumer is shown in Figure 3. The meaning of each of the steps is detailed in the following list.

1. *Placing an Order.* In this example, the customer starts off by obtaining an STR record that describes the intended transaction. The customer may obtain this STR by clicking on a banner ad, via unsolicited e-mail, or as the result of an explicit search.

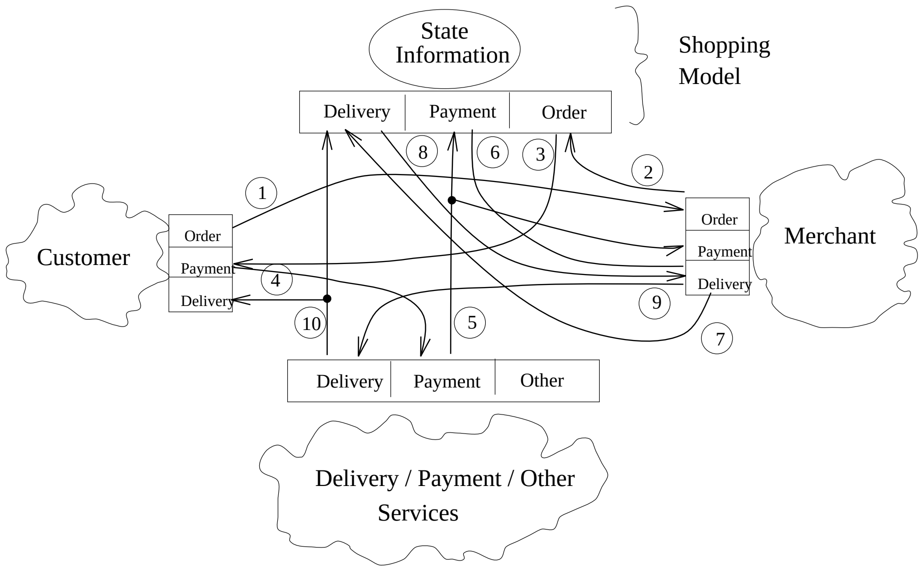

Figure 3: Steps in a subscription based service

The STR record that the customer obtains is not fully specified initially. It only contains references to the shopping model event handlers, as well as to those for the merchant. Most of the fields in the order record may already be filled in, for example, the expiration date, the goods description, the terms and conditions, and the price. Thus, this partially completed STR represents what the merchant is willing to offer, in our example, a subscription for, say, 52 weeks to a magazine.

To place the order for the subscription, the customer adds his own information (the customer's event handlers for payment and delivery) to the STR and then invokes the `PlaceOrder` method on the `MerchantOrder` object, passing it the newly completed STR (Message 1 in Figure 3). (In Section 4 we discuss how the customer may negotiate, say for a different price than what is offered.)

2. *Verifying the Order.* The `MerchantOrder` verifies the integrity of the order, checking that the customer has filled in the appropriate information (event handlers for delivery and payment) and ensures that there were no changes to the content of the order (such as the price or goods description). Finally the merchant checks that the order has not expired. Assuming these criteria are met, the `MerchantOrder` signals (Message 2) to the `ShoppingOrder` that it is appropriate to continue.

3. *Requesting Payment.* The `ShoppingOrder` receives the report containing the Status from the merchant. If the Status is `Complete`, which is a sign of a successful verification by the `MerchantOrder`, the `ShoppingOrder` knows that payment is the next step. Consequently, the `ShoppingOrder` invokes the `Pay` method on the `CustomerPayment` event handler, passing the STR as an argument (Message 3). The next two steps are similar to those illustrated in our first example in Section 2.

4. *Initiating the Payment.* The main function of the `CustomerPayment` handler is to check if payment is acceptable to the customer and to set up the payment of funds through the event handler (proxy) for a payment mechanism. Although not shown in the figure, the `CustomerPayment` object may have to contact the `MerchantPayment` to obtain additional details on how the funds should be paid. The payment is initiated by Message 4. This message contains references to a list of objects that should be notified of the progress of the payment transaction. In this case, the *notification list* contains a reference to the `ShoppingPayment` object. (Who should be notified is specified by the `PaymentNotificationList` of the STR. This information goes into the notification list of Message 4. There is also an analogous `DeliveryNotificationList`.)

5. *Acknowledging Payment.* The payment mechanism proxies produce status updates, showing the eventual resolution of the payment. The shopping model's `ShoppingPayment` object receives these noti-

fications. Message 5 (actually two messages sent to different recipients) from the payment mechanism proxies indicate that payment was completed successfully. The `ShoppingPayment` object goes on to make an update to the local state of the STR, extending the customer's subscription by the number of issues just purchased, and adding him to the merchant's subscriber list (retained at the `ShoppingDelivery` handler) if necessary.

6. *Registering the* `ShoppingDelivery`. An unusual feature of subscriptions is that delivery is extended over time. The content that the subscriber is buying has not even been produced at the time of purchase. Therefore, the control over the delivery schedule has to lie outside the e-commerce application, in the hands of an editor who has the ultimate authority to say "This is the complete issue ready to ship." The `ShoppingDelivery` handler has to know when this condition is met. Therefore, it calls the `Register` method on the `MerchantDelivery` handler (Message 6), so messages of this variety will be passed along.

7. *Announcing availability of new issue.* When the magazine editor has produced the final draft of a week's copy and informs the `MerchantDelivery` handler, it in turn broadcasts the message to all objects which have previously invoked the `register` method. In this case, it results in invoking a subscription-specific method `SendIssue` (Message 7) on the ShoppingDelivery.

8. *Sending Issues.* This method results in the invocation of the `SendGoods` method on the Merchant-Delivery handler for each STR (Message 8). The editor repeats the `SendIssue` call on the `ShoppingDelivery` each week when the new issue becomes available, relying on the shopping model to make sure that all and only paid subscribers receive it.

9. *Initiating Delivery.* The `MerchantDelivery` object is responsible for sending the goods to the customer. However, as in the case of payment, the handler does not perform the actual low level delivery protocol, but merely makes a high level call (Message 9) on the `ServiceDelivery`, a proxy driving the low level protocol, say Cryptolope in this example.

10. *Acknowledging Delivery.* As each issue is successfully delivered, the `ShoppingDelivery` is notified (Message 10) and decrements the number of issues left in the subscription, as recorded in the local state of the STR. Upon successful conclusion of the subscriber's last issue, the subscriber is removed from the active list maintained in the `ShoppingDelivery`.

3.2 Session

The session model is applicable when a number of requests occur within the same billing cycle. As in the case of the Knight-Ridder Information Service, there are several database requests which are made (order placed) and answered (goods delivered) before the monthly bill is received (aggregate payment made). Therefore, the session shopping model has one shopping model which may be summarized as (Order, Deliver) until 30 days; Pay.

The customer receives the STR when he signs up with Knight-Ridder, and gets a login identity. This could be stored in the STR's local state, or as part of a customer application. When the customer desires to make a service request within the session, he uses the STR (with the appropriate customer event handler information included), and calls the `PlaceOrder` method of the `MerchantOrder` (Figure 4, Message 1). The `MerchantOrder` verifies that it is still a legitimate order (for instance, the login has not been reported as compromised), and notifies the `ShoppingOrder` (Message 2). This handler sets a local timer to go off when the session expires (in 30 days in our example), and then calls the `MerchantDelivery` (Message 3). That event handler causes the service to be delivered (Message 4). As a by-product, the delivery mechanism proxies notify the `ShoppingDelivery` that delivery has been performed successfully (Message 5).

When the timer at the shopping model goes off, the `ShoppingPayment` is activated, and it invokes the `Pay` method on the `CustomerPayment` handler (Message 6). If the `CustomerPayment` handler arranges for payment (Message 7), the payment level proxies will notify the `ShoppingPayment` of the outcome (Message 8). If the payment was successful, the shopping model reverts to permitting service requests followed by an aggregate monthly payment. If the payment is not successful, `ShoppingPayment` is also notified, and marks the session as inactive.

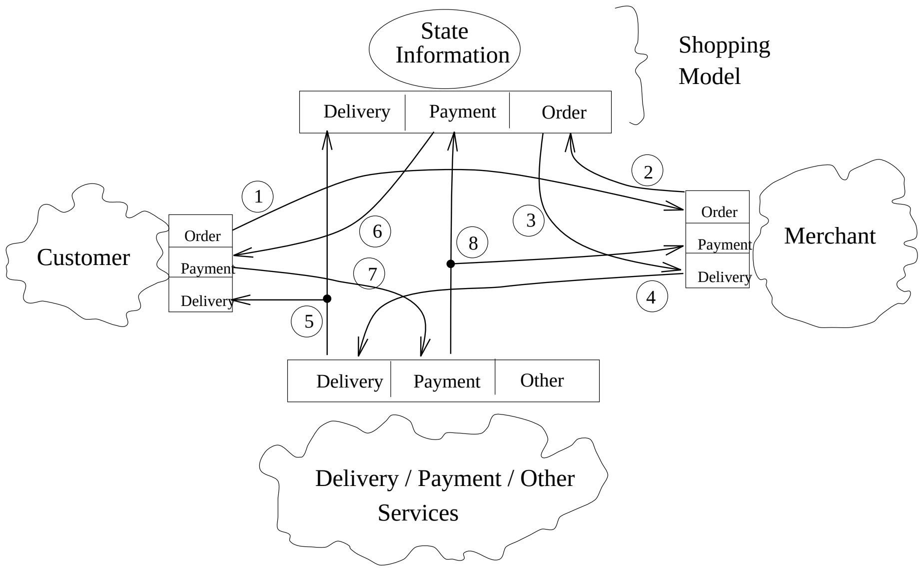

Figure 4: Steps in a session based service

3.3 Shareware

In a shareware transaction, there is not agreement about the shopping model and order before the transaction begins. The merchant simply sends out the goods, hoping that a high enough percentage of the recipients will become registered users or send payment so that the cost of the delivery is justified. Therefore, in this case, the merchant starts the process by sending the goods to the customer, via the delivery proxies (Messages 1 and 2 of Figure 5).

The delivery includes both the goods and the STR forming the context under which the merchant presents the goods. The order form may include terms and conditions the merchant hopes the recipients will respect, and consumers who wish to continue the transaction may do so by following the STR. In our example, the reader does in fact choose to provide compensation for the newsletter. The customer invokes the `Pay` method on his own `CustomerPayment` object, which results in the funds being sent (Message 3). When Message 3 is sent, it includes the payment notification list obtained from the STR. In this case, the list contains not just the `ShoppingPayment` object, but also the `MerchantPayment` object. This is because the merchant wants to track the payments, say to track the effectiveness of the sales promotion. When the `ShoppingPayment` object receives notification of successful payment (Message 4), it invokes the `MerchantDelivery` handler (Message 5) to send a further delivery, perhaps a thank you letter, (Message 6), whose receipt is acknowledged by the delivery mechanism proxy as a status report to the `ShoppingDelivery` object (Message 7).

3.4 Other Shopping Models

The examples presented here are but a few points in a rich space of options. They demonstrate the range of flexibility of the shopping model concept, and show how the event handlers of the customer, merchant, services and shopping model can work together to generate a broad range of complex behaviors.

4 Negotiation

In the previous examples, there was always agreement between the merchant and consumer on all the terms of the sale. It is more realistic to recognize that there can be disagreements at two levels: over the order, and over the shopping model itself. The methods for resolving these differences are described in the next sub-sections.

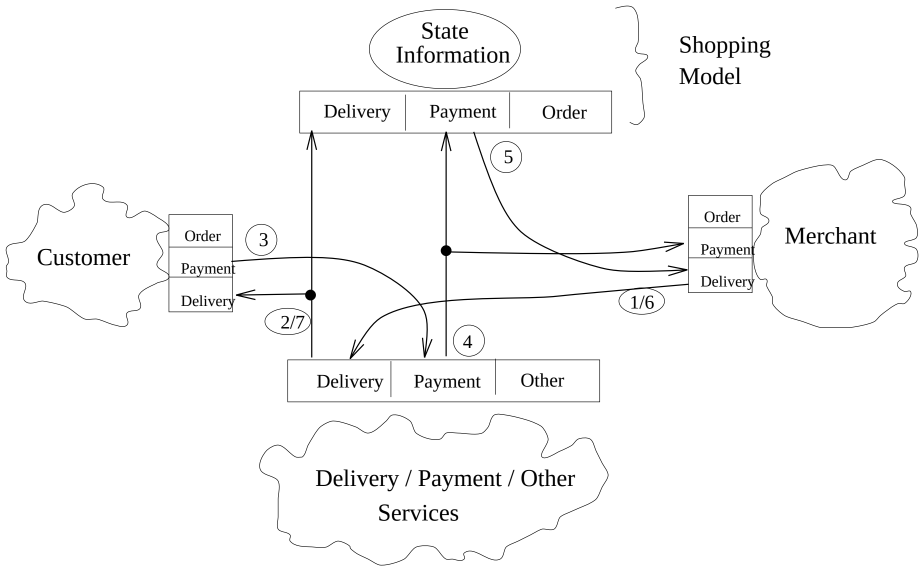

Figure 5: Steps in a shareware based service

4.1 Negotiating the Order

The standard notions of negotiation fall within the scope of negotiating the order. For instance, the consumer may try to get a lower price by negotiating the order. Orders also contain the terms and conditions, so if a consumer wished to acquire rights not only for personal use but also for three public performances, then she would undertake order negotiation. Also, modifications to the planned delivery or payment mechanism can be negotiated through this process. Of course, negotiations might include several of these aspects at once, offering, for instance, a lower price if cash is used rather than a credit card.

The actual negotiation process, including the sequence of orders and counter-offers is not specified by the protocol. Instead, only the mechanisms for the negotiation are defined. If a customer receives an order form and wishes to negotiate one of the parameters from it, he constructs an alternative which would be acceptable. He may also fill in the optional field called `CounterOffers` which gives the merchant further guidance on how to adjust some aspect of the order form. The `CounterOffers` field is a list of `Order` records, which are suggested alternatives to the current order form. Either the merchant or consumer may propose counter-offers.

In Figure 6, one round of negotiation is added on to the session example from Figure 4. Once the customer constructs the counter-offer, it is passed to the `MerchantOrder` (Figure 6, Message 1a) that was specified in the STR associated with the order form. There may be a large number of messages (but in the figure, just Message 1b) going back and forth between the `MerchantOrder` and `CustomerOrder`. Ultimately, the `MerchantOrder` is called (Message 1c) with an acceptable order. The remaining steps of the session model (Messages 2 – 8) are unaffected. In the event that no negotiated settlement can be found, either side can call off the process by sending a “failure” message to the `ShoppingOrder`.

A similar process works when the merchant receives an order which he considers unacceptable. The `MerchantOrder` creates an alternative counter-offer, using application knowledge plus additional information about the consumer's request contained in the `CounterOffers` field. Once the counter-offer is completed, it is sent to the `CustomerOrder`, for approval or further discussion.

4.2 Negotiating the Shopping Model

A second type of negotiation opens up more of the shopping process for discussion. In negotiations, one party may say “I prefer unlimited access for six months” while the other party offers only pay-per-view. The process of proposing, revising, and ultimately accepting or rejecting the proposals is identical to that followed by the negotiations of orders. The order form includes a reference to the STR which contains the

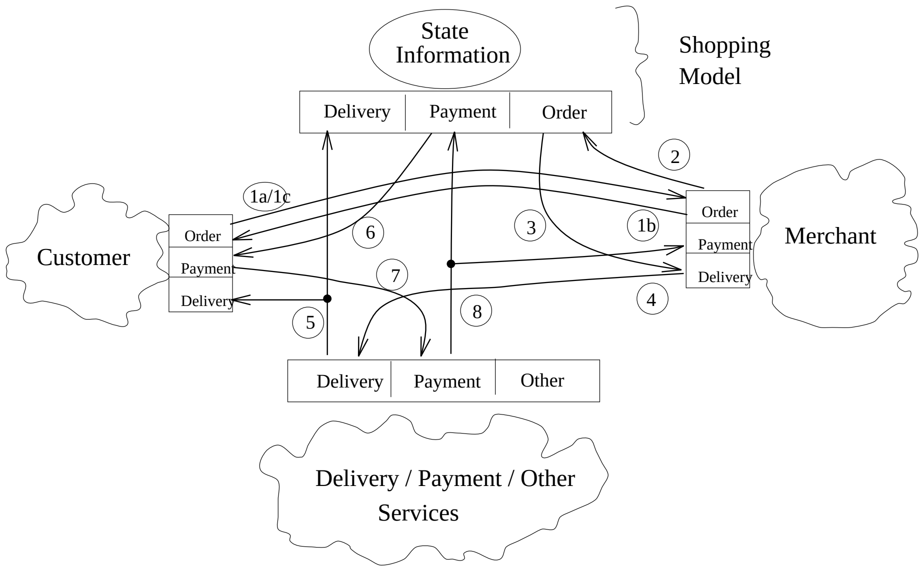

Figure 6: Session shopping model with initial negotiation stage

fields specifying the `ShoppingOrder`, `ShoppingPayment`, and `ShoppingDelivery`, together comprising the shopping model. Therefore, the negotiation over which shopping model to use is the same as that for any other field of the order, e.g. the price or terms and conditions. This approach does presuppose, however, that both customer and merchant are able to determine whether a newly proposed shopping model is acceptable. The parties may be able to extract a meaningful understanding of the sequence of steps imposed by any given shopping model, or they may limit alternatives to those in a library provided by a trusted party such as the Better Business Bureau. Since the shopping model itself is independent of the merchant's business process or content inventory, many merchants may use the same implementation of a shopping model. Therefore, it would be possible to create and widely distribute a library of shopping models.

4.3 Pull model

The previous examples have dealt primarily with the cases where the consumer is buying something that the merchant has advertised for sale. An alternative model is where the customer first proposes the order and the shopping model. So, for instance, if the consumer has an information request, and wants to get some non-standard information from the merchant, this alternative "pull" model may be appropriate.

First, the customer creates the order defining the goods, price, and terms and conditions that will go into the transaction. Second, the customer finds an appropriate shopping model, which may be one of the merchant's or perhaps one from a third party provider. The shopping model reflects the proposed sequencing of order, deliver, and payment steps, along with any iterations among those steps. The customer also creates an STR with the customer event handlers filled in. The customer sends this STR (with its pointer to the order record) to the `MerchantOrder`, which determines whether the customer's proposal is worth fulfillment or negotiation.

If the merchant decides to accept the order and shopping model as proposed, it fills in the merchant event handlers. Negotiation over price or other terms, including the shopping model may proceed as described above. It is irrelevant to the negotiation which party (merchant or customer) began the process.

If the `MerchantOrder` can agree to the terms of the order and shopping model, then it sends a `Complete` message to the `ShoppingOrder` handler of the newly agreed upon shopping model. From there, the transaction proceeds just as if the order and shopping model had been originally proposed by the merchant.

5 Connection to U-PAI and U-DEL

In previous work [5], we defined U-PAI, a Universal Payment Application Interface. The purpose of U-PAI was to insulate an application from the differences in payment mechanisms, providing a higher level interface. Proxies for each payment mechanism translate from that abstract interface to the native protocols of the payment mechanism. This treatment of payments was one of the building blocks for the current system. For example, the `ServicePayment` handler corresponds to a U-PAI *AccountHandle*, or proxy to a user's account with a particular payment mechanism. Referring to the subscription example of Figure 3, Messages 4 and 5 are directly using the U-PAI protocol. Similarly, the `CustomerPayment`, `MerchantPayment`, and `ShoppingPayment` all support the interface for a U-PAI *Monitor* object which receives the status reports for a payment.

In [3] we argue that "bits are bits" and there is nothing intrinsically different about the bits of "content" from those representing "payment". Therefore, we define a protocol called U-DEL for handling different delivery mechanisms based on an analogy to payment. Therefore, the `MerchantDelivery`, `ShoppingDelivery`, and `CustomerDelivery` are U-DEL Monitor objects which make calls on U-DEL *DeliveryHandles*.

6 Opening for Business

In this section we consider the steps needed to launch an electronic document business using the shopping model framework. With one exception (Step 2), the process is independent of the particular shopping model. In the case where the shopping model is relevant, we will use the pre-paid subscription as an example. By completing the following steps, the merchant can handle both the pre-paid and trial subscription shopping models. Note that all of these steps are done without interaction with the customers.

1. *Define Merchant Event Handlers.* The merchant has to develop three event handlers: `MerchantOrder`, `MerchantPayment`, and `MerchantDelivery`. These handlers do not depend on the shopping model.

   The `MerchantOrder` receives and processes incoming orders. In this example, the `MerchantOrder` is also empowered to negotiate along two dimensions. First, if the customer asks for a "Trial Subscription", the `MerchantOrder` can permit that. Second, the `MerchantOrder` can offer the six, twenty-four, or thirty-six month subscriptions at the appropriate rate. Note that these steps are not dependent on the shopping model. To the `MerchantOrder`, they are merely "Offer 1" and "Offer 2", with nothing specific to subscriptions.

   The `MerchantPayment` handler must receive payments from the customers. In this example, it does that by managing U-PAI `AccountHandles` for the particular payment mechanisms. The `MerchantPayment` handler must also be able to work with the `CustomerPayment` to select the appropriate payment mechanism to use.

   The `MerchantDelivery` handler must coordinate the delivery of the goods that the customer ordered. As in the payment case, this job is handled by using a subsidiary protocol for delivery, U-DEL, to manage the operation of a `DeliveryHandle` for the Cryptolope mechanism.

2. *Define Acceptable Shopping Models.* In many cases, the merchant will just be able to use a standard shopping model from a library of common ones. Here, we go through the exercise of sketching out from scratch a shopping model for pre-paid subscriptions. Each shopping model is composed of pointers to three event handlers whose behavior taken together form a "script" for the transaction.

   The order event handler is a `ShoppingOrder` object, described formally in ISL in the Appendix. Like its payment and delivery counterparts, it needs to support only one method, `Notify`. The `ShoppingOrder` determines what to do next, based on whether the order was successful or not. In the pre-paid subscription example, a completed order is followed by billing the customer. (In the following code sections, we use pseudo-code resembling Python, but any language could be used to implement the event handlers.)

   ```
   class ShoppingOrder():
   ```

```
  def notify(STR s, Status stat)
    # Check to see if order has been received
    if stat.MajorStatus == Complete:
       s.CustomerPayment.Pay(s)
    elsif: stat.MajorStatus == Failed:
       delete s  # remove record of aborted transaction
```

The `Status` type consists of two fields: a "MajorStatus" and "MinorStatus". The "MajorStatus" provides only a very coarse distinction of whether the payment has successfully completed, is on-going, or failed. By conflating all the possible responses into these three categories, we enable the shopping application to make a reasonable response even if it is unfamiliar with the details of the case. On the other hand, the shopping model does have access to the detailed reason for failure in the "MinorStatus" field. Therefore, it can attempt a very targeted repair effort enabling the transaction to continue.

The second event handler is a `ShoppingPayment`. In the pre-paid subscription, the payment completion is a pre-condition for delivery of the goods. If the first payment attempt from the customer fails, he should be permitted to present an alternative form of payment. The U-PAI system uses a `PaymentControlRecord (PCR)` to maintain the details for a payment. It is generated by the U-PAI `AccountHandles` and permits additional payment related functions such as aborting the payment or getting its status.

The `ShoppingPayment` handler defined here is a U-PAI `Monitor` object, which receives an updated status for a particular PCR. One field of the PCR gives the context of the payment, which is set to be the STR. If the payment completed successfully, the new subscriber is added to the active list maintained by `ShoppingDelivery`, and the number of issues (stored in the local state of the STR) is also increased.

```
class ShoppingPayment():

  def notify(UPAI.PaymentControlRecord PCR, Status stat):
    str = PCR.GetContextID()  # ShoppingTransRec is context for payment
    if stat.MajorStatus == Complete:
        str.ShoppingDelivery.AddSubscriber(str)
# Add this new subscriber to list maintained by shopping model
        str.localstate.SubscriberIssues += str.Order.Goods.NumIssues
# increase the number of issues remaining by the number paid for
    elsif stat.MajorStatus == Failed &&
          stat.MinorStatus == NotSufficientFunds:
        str.CustomerPayment.SelectPaymentMechanism(str)
```

The third event handler is a `ShoppingDelivery`. In the pre-paid subscription example, this object is responsible for decrementing the number of remaining issues each time a new one is sent, and removing the subscriber from circulation when the subscription runs out.

```
class ShoppingDelivery():

  def notify(UDEL.DeliveryControlRecord DCR, Status stat):
    if stat.MajorStatus == Complete:
      str = DCR.GetContextID()  # ShoppingTransRec is context for delivery
      str.localstate.SubscriberIssues -= 1
      if str.localstate.SubscriberIssues == 0:
          self.removeSubscriber(str)

  def addSubscriber(STR str):
    self.SubscriberList.add(str)
```

```
def removeSubscriber(STR str):
  self.SubscriberList.delete(str)

def SendIssue(IAny.Any issue):
  for str in self.SubscriberList:
    str.MerchantDelivery.SendGoods(issue, str)
```

The merchant would go through a similar process to create other shopping models (such as the one offering a trial subscription), or would acquire them from code libraries.

3. *Create the list of available Shopping Models.* Once the different candidate shopping models have been created, they can be added to the list of shopping models that the merchant is willing to offer. This list is stored as part of the merchant's application.

4. *Enable the Order Taking Process.* This simple step (invoking the `Start` method on each `MerchantOrder` object) requires the merchant to explicitly acknowledge when he is "open for business".

5. *Distribute Order Forms.* In the subscription model, there is a means to let a customer know that he or she is eligible for a subscription to the magazine at the given rate. The order form which a merchant distributes to the possible buyers is the STR with the merchant event handlers filled in and the included pointer to an order record which contains the goods offered, price, terms and conditions, etc. This order form may be distributed through advertisements or sent to the `CustomerOrder` event handler. Alternatively, a banner advertisement may allow the consumer to download an order form.

7 Security and Trust

Although problems of security are critical for applications where money and intellectual property are being transferred, these considerations are outside the scope of this paper. A complete treatment of the security issues is grounds for future work, and we believe that the mechanisms to implement a reasonable level of security are being developed and could be integrated into this framework. In particular, we assume that there will be widespread use of access control on a method-by-method basis for computational objects. In addition, since we build upon existing mechanisms for payment and delivery, this system inherits some of their security. Additional work is necessary to ensure that the security properties of these mechanisms are retained for the rest of the system.

In addition to concerns about attacks from outsiders and eavesdroppers, there is also a legitimate concern that a rogue merchant or customer will attempt to defraud the other. For instance, a merchant might implement a series of processes that claims to charge one price but actually charges another. While a series of checks and balances implemented by the customer can guard against many such traps, a broader solution may be required. One option would be to have a trusted third party provide signed implementations of the basic objects to implement common interaction models. Although the customer would still need to ascertain that the parameters such as pricing were appropriate, there would be a greater trust of the underlying model.

8 Conclusion

We have introduced the shopping model framework, an API to facilitate the creation of commerce applications. The use of proxies for payment and delivery mechanisms enables a plug-'n'-play simplicity to these external services. The major contribution, however, is the distillation of the event handler environment of merchants, customers, and especially the shopping models. By separating these components, we enable a single shopping model to be used by multiple merchants, and a single merchant to use multiple shopping models.

We showed the range of flexibility of the shopping model framework by providing examples of five different types of interactions, and demonstrated that shopping models could implement them, without requiring changes to the merchant or customer code.

We plan to implement this framework within the Stanford Digital Library project. We have already shown how U-PAI provides independence from payment mechanisms. From that starting point, we plan to implement the shopping model system to provide great flexibility in the types of commercial transactions permitted in the digital library. We also plan to extend this work by resolving the security issues to protect the money, intellectual property, and privacy of all the participants to the transaction.

Acknowledgments The authors wish to thank the members of the Stanford Digital Library Project Economics Issues Group for helpful discussions. Special thanks to Steve Cousins for his comments.

References

[1] Steve B. Cousins, Steven P. Ketchpel, Andreas Paepcke, Hector Garcia-Molina, Scott W. Hassan, and Martin Röscheisen. InterPay: Managing multiple payment mechanisms in digital libraries. In *DL '95 proceedings*, 1995.

[2] DigiCash. DigiCash brouchure. Available at http://www.digicash.com/publish/digibro.html, 1994.

[3] Stanford Digital Libraries Economic Issues Core Group. Bits are bits, so delivery = payment. Technical Report, Stanford Digital Library Project, 1996.

[4] IBM infoMarket Development. IBM cryptolope containers. Technical Report, IBM Corp., http://www.infomkt.ibm.com/ht3/crypto.htm, Nov 1995.

[5] Steven P. Ketchpel, Hector Garcia-Molina, Andreas Paepcke, Scott Hassan, and Steve Cousins. UPAI: A univeral payment application interface. In *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996.

[6] Olin Sibert, David Bernstein, and David Van Wie. Securing the content, not the wire, for information commerce. Technical Report, InterTrust Technologies Corp., http://www.intertrust.com/architecture/stc.html, 1996.

[7] L.H. Stein, E.A. Stefferud, N.S. Borenstein, and M.T. Rose. The Green commerce model. Technical report, First Virtual Holdings Incorporated, October 1994. Available at http://www.fv.com/tech/green-model.html.

9 Appendix

```
INTERFACE ShoppingModels  (* Version 1.0.  For current version see:
                  http://www-diglib.stanford.edu/diglib/software/SM.isl *)
  IMPORTS
    IAny,  (* See: http://www-diglib.stanford.edu/diglib/software/IAny.isl *)
    CosPropertyService,
       (* See: http://www-diglib.stanford.edu/diglib/software/CosProp.isl *)
    UPAI (* See: http://www-diglib.stanford.edu/diglib/software/UPAI.isl *)
  END;

TYPE String = ilu.CString;

TYPE RefIDType = String;

TYPE STR = RECORD

  STRID : RefIDType;
  STRStatus : StatusHistory;

  ShoppingOrder : ShoppingOrderType;
  ShoppingDelivery : UDEL.Monitor;
  ShoppingPayment : UPAI.Monitor;
  Description : String;

  CustomerOrder : CustomerOrderType;
  CustomerDelivery : CustomerDeliveryType;
  CustomerPayment : CustomerPaymentType;
  MerchantOrder : MerchantOrderType;
  MerchantDelivery : MerchantDeliveryType;
  MerchantPayment : MerchantPaymentType;

  PaymentNotificationList : UPAI.MonitorList;
  DeliveryNotificationList: UDEL.MonitorList;

  LocalState : IAny.Any;

  OrderRef : Order;

 END;


TYPE OrderList = SEQUENCE OF Order;

TYPE Order = RECORD

  RefID : RefIDType;
  RefSTR : STR;
  Expiration : Time;

  GoodsDescription: String;
  Price : UPAI.Amount;
  TermsandConditions: IAny.Any;

  PaymentMechanismOptions : UPAI.AccountHandleList;
  DeliveryMechanismOptions: UDEL.DeliveryHandleList;

  CounterOffers:  OrderList;

END;

TYPE MajorType = ENUMERATION
         Complete,
         InProgress,
```

```
         Failed
    END;

TYPE Status = RECORD
        MajorStatus : MajorType,
        MinorStatus : IAny.Any

  (* Typical values are strings:

      Merchant unable to provide goods,
      Merchant unwilling to meet Order,
      payment failure,
      delivery failure, ....*)

    END;

TYPE StatusHistory = SEQUENCE OF Status;

TYPE ShoppingOrderType = OBJECT
  METHODS

    Notify(what : STR, status : Status)
      (* Notify informs shopping model of new status of Order  *)
  END;

(*-----------------------------------------------------------------------*)
(*--                           CUSTOMER METHODS                        --*)
(*-----------------------------------------------------------------------*)

TYPE CustomerOrderType = OBJECT
    METHODS
       PlaceOrder(s : STR),
       ReceiveOrderForm(s : STR)
    END;

TYPE CustomerDeliveryType = OBJECT
    METHODS
       ChooseDeliveryMethod(s : STR, d : DeliveryHandleList) : DeliveryHandle,
        (* Selects the preferred delivery method from list *)
       ReceiveGoods(s : STR)
        (* Prepares to receive via U-DEL *)
    END;

TYPE CustomerPaymentType = OBJECT
    METHODS
        ChoosePaymentMethod(s : STR, al : AccountHandleList) : AccountHandle,
         (* Selects the preferred payment method from a list *)
        Pay(s : STR)
         (* Makes UPAI calls *)
    END;

(*-----------------------------------------------------------------------*)
(*--                           MERCHANT METHODS                        --*)
(*-----------------------------------------------------------------------*)

TYPE MerchantOrderType = OBJECT
     METHODS
        Start(),
        (* Called by Merchant to enable the order taking process *)
        PlaceOrder(s : STR),
        (* Can ignore orders, invoke SCR, or try negotiator *)
        VerifyOrder(s : STR)
        (* Determines that this order is completed, unexpired, correct *)
     END;
```

```
TYPE MerchantDeliveryType = OBJECT
    METHODS
       ChooseDeliveryMethod(s : STR, d : DeliveryHandleList) : DeliveryHandle,
        (* Selects the preferred delivery method from list *)
       SendGoods(goods: IAny.Any, s : STR),
        (* Invokes U-del *)
       Register(s : STR, handlers : INTEGER)
        (* Causes the Shopping Handlers encoded by the argument (as a
           bit vector 1 = order, 2 = delivery, 4 = payment), to
           receive status updates. *)

    END;

TYPE MerchantPaymentType = OBJECT
    METHODS
        ChoosePaymentMethod(s : STR, al : AccountHandleList) : AccountHandle,
         (* Selects the preferred payment method from a list *)
        ReceivePayment(s : STR)
         (* Prepares receive, is the UPAI application that signals SCR on
            Completion. *)

    END;
```