U-PAI: A Universal Payment Application Interface [*]

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

Abstract

The progress of electronic commerce has been stymied by the lack of widely accepted network payment mechanisms. A number of proposals have been put forward, and each one offers a slightly different protocol and set of features. Yet none has achieved the critical mass to become an accepted standard. We believe that there will continue to be a variety of payment mechanisms, so in this paper we propose U-PAI, a universal payment application interface that will enable a programmer to write for one interface, and then interact with any payment mechanism. Each payment mechanism can support the universal API directly, or a *proxy* or wrapper can be built to translate U-PAI calls to the appropriate native calls supported by the payment mechanisms. In this paper we illustrate how two such proxies could be built. We also provide, in the appendix, a full CORBA specification of U-PAI.

1 Introduction

A payment mechanism is a means by which economic value is transferred between two parties, possibly using some intermediaries. It should be secure, easy to use, and have low transaction costs. Even though all electronic payment mechanisms have these same goals, there are many variations between mechanisms, (see for example, [9, 4, 7, 6]). Some of the variations can be minor, e.g., the order or nature of parameters in a function call. Other differences are more substantial, such as using different transport mechanisms and protocols like HTTP, telnet or e-mail. The most significant difference, however, is the order of steps required to execute a payment. One payment mechanism, Millicent[7], requires the payer to acquire "scrip" from a broker before an interaction, while a second, the anonymous credit card[6], channels all communications through a re-mailer to keep identities hidden. If a merchant wants to support several payment mechanisms, not only must the merchant have accounts with each, but he or she must also tailor the application software to determine which mechanism is in use by the customer and generate the proper payment protocol steps to the customer and intermediaries.

The diversity of payment mechanisms may be beneficial in the long run because it encourages competition and enables an exploration of a broader space of solutions. However, this diversity is also a significant barrier to commerce: customers must maintain accounts with several different payment mechanisms. Furthermore, merchants and customers both find that there is no standard way for payment mechanisms to interact with application software such as a browser or electronic storefront.

Our goal in this paper is not to add to the diversity by introducing another payment mechanism, but rather to define a common set of functions that act as a layer of abstraction between application software and payment mechanisms. This Universal Payment Application Interface (U-PAI) will ease the burden on software developers at both the consumer and merchant level. Merchants and customers do not need to customize their applications to support each individual payment mechanism, since an application supporting this one universal API will interact with a broad range of payment mechanisms.

We hope that the benefits of standardization will encourage payment mechanism providers to support U-PAI (perhaps in addition to their own API which provides additional or different functionality). However, we recognize that payment systems providers see their proprietary protocols as a differentiating fac-

[*]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.

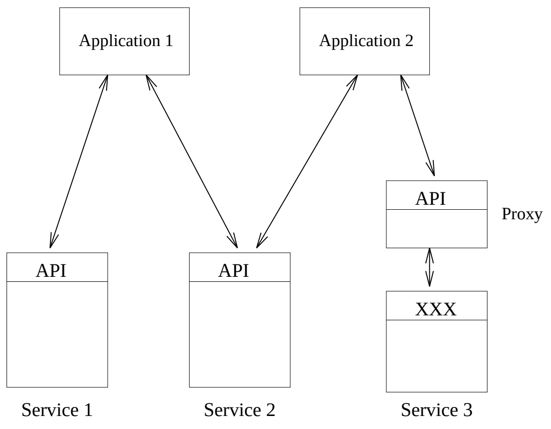

Figure 1: Universal Payment Application Interface abstracts payment mechanism internals

tor and way to retain market share. One approach to achieve widespread use of the U-PAI protocol would be to propose it to the relevant standards bodies and proceed through the ratification process.

An alternative approach is to build *proxies* or wrappers or gateways to popular payment mechanisms, as illustrated in Figure 1. Each proxy translates U-PAI calls into native calls to the underlying payment mechanism. This notion of proxy is widely used when accessing heterogeneous resources, be they databases, search engines, or other services [8]. By developing proxies and distributing them freely for the most popular payment mechanisms, we can encourage application developers to experience the benefits of using a single protocol, which may result in their reluctance to devote implementation effort to systems which do not support the protocol. This type of pressure may effectively encourage other payment systems providers to support the interface.

Of course, since each payment mechanism offers different features, it is impossible for a single API to capture all of the functionality of all of the mechanisms. Thus, the challenge we face in designing the common API is to identify the essential features that are used in the vast majority of interactions. A second challenge is to design, for these common features, an elegant interface that simplifies the programming task. One significant aspect of this challenge is that important steps need to be asynchronous, non-blocking calls. Asynchrony permits multiple payments to be in process at the same time, or may allow a payment to be aborted after it has been authorized, but before it has been completed. We believe that U-PAI meets those goals.

Having defined U-PAI, the next challenge is to show that one can build proxies to existing payment mechanisms, and that these proxies can support the necessary common functionality even if the underlying mechanism uses a different payment model or ordering of steps. We have studied a number of existing mechanisms and shown how U-PAI can support the basic functionality of all these schemes. We will illustrate two such proxies, one supporting First Virtual, the second, DigiCash's ecash product.

In the following section we describe some related work. Section 3 defines all of the functions which are part of the interface. Section 4 shows a sample transaction, giving each step from start to finish. Section 5 shows how a proxy might be constructed for the First Virtual payment mechanism. An equivalent ecash proxy appears in Section 6. In Section 7, the case of failed transactions is considered in greater detail, along with security concerns. Finally, Section 8 offers a summary. The full CORBA specification in ISL, the interface specification language of Xerox PARC's ILU, appears in the Appendix.

2 Related Work

It is important to note that payment mechanisms and U-PAI are only one part of a larger electronic commerce environment. U-PAI only covers the basic functionality of accounts and payments, e.g., checking the balance of an account or transferring funds from one account to another. It does not cover price negotiation, return of defective goods, bidding, and other commerce issues. These require other API's that would work in conjunction with U-PAI. A broader view of the issues related to electronic payments may be found in [1], which presents the Generic Electronic Payment Services framework. Of the five modules discussed there, U-PAI performs "Capability Management" tasks, some of which may appear in the higher level "Payment Interface Manager".

Another attempt to address the diversity of payment mechanisms is the Joint Electronic Payment Initiative (JEPI) project, co-sponsored by CommerceNet and W3C. The focus of JEPI is reaching agreement between the customer and merchant on a payment mechanism[2]. JEPI is built on top of Eastlake's Universal Payment Preamble (UPP)[5]. Neither of these systems achieves the level of integration that is proposed in U-PAI. Application developers must still implement a different protocol for all of the payment mechanisms they choose to support; JEPI and UPP merely allow the customer and merchant to select the protocol that a particular transaction will use. Therefore, it would be possible to employ JEPI to select a payment mechanism and then use U-PAI to control the processing within that mechanism.

Finally, we note that our work is being done in the context of the Stanford Digital Libraries Project, where we are studying how to provide access to the resources and services being developed under the NSF/DARPA/NASA Digital Libraries Initiative (see `http://www.dlib.org/projects.html`). Clearly, payment is one of the central issues in such an environment. This work was performed in collaboration with EIT/VeriFone, under the auspices of CommerceNet (see `http://www.commerce.net/`), where again, facilitating interactions among customers and merchants with different payment mechanisms is crucial.

3 API Definition

U-PAI was designed from an object-oriented point of view. The interface offers a set of active objects, with their associated methods. Making a call to U-PAI involves calling a method on one of these objects. Similarly, the interface specifies certain objects that the application is expected to have that can be called by U-PAI, for example, to notify the application when a payment transaction terminates.

The equivalent functionality of the interface can be captured through non-object-oriented means as well. Entities which are objects in the API would be construed as records, with the object ID representing an identifying index for the record. Method invocations are replaced with a remote procedure call that passes the record which is to be acted upon as an explicit parameter of the call. In the interest of clear presentation, the object-oriented method will be used throughout the rest of the exposition.

In this section we describe the main object types in the API and their methods. Some of the methods are used to access what conceptually are "internal fields" of the object. For example, as we will see later, a payment control record (`PCR`) named `P` has an `Amount` field that gives the amount of money being paid. This value can be read by invoking `P.GetAmount()` and can be set by `P.SetAmount()`. In reality, `P` may not have a field with this value (in which case we say it is not *materialized*), but `P.GetAmount()` may invoke a function to compute the amount based on other internal or external information. However, for understanding the interface, it is useful to think of `Amount` as a field in `P`. Also, note that often the `Set` method will be disabled for some fields, e.g., the application may not set the balance of an account. Thus, to describe each of our objects, we first define their "fields" and then other methods they may have. (Full formal definitions may be found in the Appendix.)

3.1 Account Handles

An *AccountHandle* instance is a representation of a real-world account. For example, a user may have several VISA `AccountHandles`, corresponding to the cards issued by different merchant banks belonging to the VISA network. The user creates an `AccountHandle` when he wishes to start making electronic payments with the account. He may query balance and credit limits on the account by making appropriate calls on the `AccountHandle` object.

A helpful analogy to clarify the notion of accounts and `AccountHandles` is that of UNIX files (see Figure 2). A file can be created and deleted, which corresponds to the creation and closing of a real world account. When the file exists, it is possible for a program to reference it by opening the file, making read and write accesses to it, and closing the file. In the payments world, this corresponds to generating an `AccountHandle`, making transfers, and erasing the `AccountHandle`. The real world account continues to exist even after the electronic `AccountHandle` representation has been deleted, just as a UNIX file exists after a program referencing it closes the file and deletes the file handle.

Conceptually, an `AccountHandle, ah,` has the following internal fields, although as noted below, some of them may not be actually materialized.

- `Balance:` This is the amount of available money available at `ah`'s account for payments. A positive amount indicates that the account holder has a positive stored balance. For example, DigiCash's ecash would be represented as a positive balance, since the user has already "purchased" the ecash. In contrast, a negative amount indicates the account owner owes money. Charges against a credit card would result in a negative balance that would be brought (closer) to zero when a payment was made to the card issuer. A query to this field, via `ah.GetBalance()`, will often require a real-time query to the account issuer, such as the bank that issued a MasterCard, in order to determine if non-electronic payments have been made. (In this case, we say that `Balance` is not materialized at `ah`.)

- `CreditLimit:` This is the amount of credit that may be charged on a credit-based payment mechanism. It is a negative value, and the balance on an account should never go below it. For non-credit instruments, its value is zero.

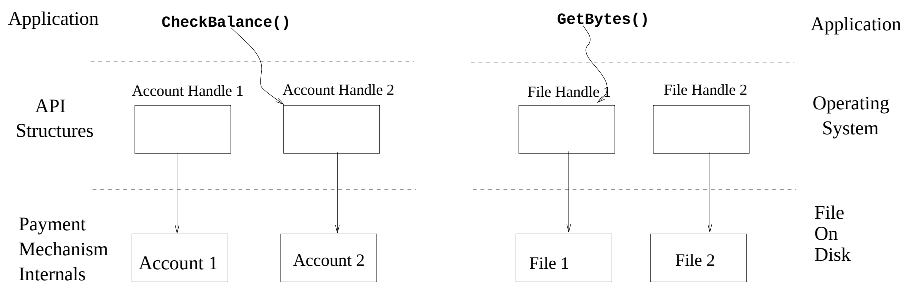

Figure 2: The similarities between an account handle and a file handle

- `AccountType`: This is an identifier (ID) of the type of account, e.g., First Virtual, VISA/SET, DigiCash, and so on. The value is of type `AccountTypeID`.

- `TransferAccountTypesFrom`: This is a list of `AccountTypeID`'s that this account can receive transfers from. So, for instance, a Mark Twain Bank ecash account can receive transfers either from another ecash account, or from the account holder's checking account.

- `TransferAccountTypesTo`: This is a list of `AccountTypeID`'s to which this account can make transfers.

- `MechanismProperties`: This is a *property set* that includes descriptive traits of the payment mechanism used by this account. Each entry in the property set is a property name and value. These properties assist the user in choosing which payment mechanism to use for a particular transaction. The name of the payment mechanism is stored in the string property `name`. The `fixed-cost` property is an amount which describes the fixed portion of the overhead cost for using this payment mechanism. The `percentage-fee` property records the variable cost. The expected time for one payment may be found in the `time` property. The boolean property `anonymous` records whether payments made using this mechanism may be linked to the user. Any other property may be added at the discretion of the payment system provider.

`AccountHandles` are typically subclassed with the specific type of the payment mechanism. The interface is inherited from the base class `AccountHandle`, but the methods are overridden with the specific details appropriate for that payment mechanism. For example, if a user wanted to create an `AccountHandle` for his First Virtual (FV) account, he would create an instance of a `FVAccountHandle`, which is in turn a subclass of `AccountHandle`. `FVAccountHandle` must have all the methods of an `AccountHandle` (though they will be implemented in a way idiosyncratic to First Virtual).

The following methods can also be invoked on an `AccountHandle, ah`:

- `OpenAccount(PropertySet acctinfo): Any`
  Typically, an application first creates a new `AccountHandle` object `ah` and then invokes `OpenAccount` on it to "initialize" `ah` to identify the appropriate real world account. The `acctinfo` parameter contains the necessary information to identify the real world account. The parameter is a property set that associates arbitrary field names with different types of parameter objects. For example, if we are opening a VISA account, `accountinfo` may contain the associations "type: VISA", "account: 123-456-789" and "expiration: 03/99." If we are opening a First Virtual account, we may need "type: FV", "name: John Doe" and "email: doe@whitehouse.gov." This method will then set up the AccountHandle as indicated, for instance, it will initialize field `TransferAccountTypesFrom` to indicate what type of account this FV account can receive funds from. We stress that this process does not establish a new FV account, it merely creates a representation of an existing FV account so that it may be used for payments through U-PAI calls. The return value of opening an account may be used as a security/authorization token to allow the object creator to identify itself to the account in the future.

- `CreateAccount(PropertySet acctinfo): Any`
  This method creates a new real world account using `acctinfo` and updates the internal fields of `ah` to refer to it. Not all payment mechanisms will offer the option of creating a real world account through purely electronic means invoked

remotely by a user. The return value may be used for authorization.

- `CloseAccount()`
  This method deletes the handle `ah`. The underlying real world account is unaffected. Future references to `ah` will result in an error.

- `DeleteAccount()`
  This method deletes both the `AccountHandle` and the real world account which it represents. Again, not all payment mechanisms will support this method.

- `GetStatus(RefIDType Ref):PaymentStatus`
  This method provides direct access to the payment mechanism's records concerning a particular transaction. In the event that the `PCR` (described next) is unavailable, an alternative (though probably more expensive) entry point exists. Not all payment mechanisms will support this method.

3.2 Payment Control Records

A *Payment Control Record* (PCR) instance is a representation of a single payment transaction. An application creates a new `PCR` for each individual transfer between two accounts. The `PCR` is then the locus of control for all activities regarding that payment.

Conceptually, a `PCR, p,` has the following fields:

- `RefID:` Provides a unique identifier for this payment. The value is of type `RefIDType`.

- `ContextID:` Identifies the context for this payment. The value, of type `RefIDType`, contains application specific information such as the invoice for which this payment is being made.

- `Amount:` the amount of money that is being paid by `p`.

- `DestAccountHandle:` Identifies the account receiving the funds.

- `DestAccountAuthorization:` Conveys the authority to deposit money in the destination account.

- `SourceAccountHandle:` Identifies the account supplying the funds.

- `SourceAccountAuthorization:` Conveys the authority to withdraw money from the source account.

- `Receipts:` Information, such as a receipt or decrypting key, that is given to `p` at the start of the transfer, and should be revealed to all participants upon successful completion. The payment mechanism may add additional receipts to this field.

- `Status:` The status of `p` is a list of entries representing the history of this payment. Each entry is made up of two components, a `MajorStatus`, which takes one of three values (`PaymentComplete, InProgress,` or `Failed`), and a `MinorStatus`, which provides greater detail about the current status. The entries are ordered with the most recent appearing at the front of the list.

  Applications may make use of these values, though in many cases, the values will be payment-mechanism specific. Some sample entry values are shown in Table 1.

- `MonitorList: Monitor` objects (described in the next section) provide a way for applications to request notifications of status changes, rather than directly poll the status of `p` through the method `p.GetStatus()`. The `MonitorList` field of `p` is a list of `Monitor` objects that must be notified when the status of `p` changes.

To perform a payment, an application creates a `PCR` object, call it `p`, with the information that describes the desired operation. Then it invokes methods on the object to start or abort the payment.

- `StartTransfer()`
  This method initiates the transfer of funds in order to effect the transfer described in `PCR p`. This call is non-blocking, so that customer processing may continue even before the payment has completed. A transfer requires authorization from both account holders to withdraw funds from one account and deposit them in another. This method should be invoked only once per `PCR`.

- `TryToAbortTransfer()`
  This function attempts to abort a transaction which was previously initiated by a `StartTransfer`. The payment may have been already completed, or reached some commit point so that it is too late to abort. Feedback is given to the calling application only via the status of the `PCR` and the `Monitor` objects.

- `UpdateStatus(StatusEntry stat)`
  This last method is invoked by whatever entity

Table 1: Payment Transaction Status Values

MajorStatus	**MinorStatus**	**Description**
PaymentComplete		Money transferred from payer to payee
InProgress		Transfer started, not completed
Failed	Aborted	The payment was aborted
Failed	NotSufficientFunds	Not Sufficient Funds for payer to make payment
Failed	NoSourceAccountSelected	The AccountHandle has not yet been associated with an account by `create()` or `open()`.
Failed	UnauthorizedSourceAccount	Payer not authorized to make payments from this account
Failed	UnauthorizedDestAccount	Payer not authorized to make deposits to this account
Failed	NonExistentDestinationAccount	Payee account not recognized
Failed	UnabletoTransfertoAccountType	Payee account wrong type

is actually performing the payment transaction to report a change in status. Parameter `stat` is a `MajorStatus`, `MinorStatus` pair which is appended to `p`'s `status` field.

Incidentally, in some cases an application may wish to make more than one payment to cover a single invoice. For example, having received a bill for some delivered good, the application may wish to pay half of the amount due with a credit card and the other half with a check. In this case, the application creates two separate `PCR`s, each with the appropriate amount to pay. In this case, each record could have the same `ContextID` field since the same invoice is involved.

3.3 Monitors

A *Monitor* instance is an object used to supplement the status tracking feature of a `PCR`. Rather than requiring the application to routinely poll the status of the `PCR`, the application programmer may choose to implement a `Monitor` object which receives notifications whenever the payment mechanism updates the status of the `PCR`. Several such `Monitor` instances may be active at any time. For example, monitors acting on behalf of the payer and payee (and any other parties to the transactions, such as a state tax board) act as the recipients of messages which the `PCR` re-broadcasts as the transfer proceeds. For instance, if a bank refuses a check due to insufficient funds, the `PCR` reflects a failed status, and passes that information to each active `Monitor`. `Monitor` objects are written by the application programmer, providing the linkage between the result of the payment mechanism and the desired behavior of the application.

A `Monitor` object, `m`, conceptually has a `status` field just like a `PCR`. The following method can be invoked on `m` to update the field:

- `Notify(PCR p, StatusEntry s)`
  This function updates the record of the transaction's status as recorded at `m`. In practice, this method also performs application specific tasks, depending on the nature of the notification.

In many cases, this basic `Monitor` class will be subclassed by the application programmer to provide additional functionality. For instance, one common usage will be to provide monitors with timeout capabilities. In this case the subclass may add methods such as `register` to define a timeout and `unregister` to cancel it. If a timeout occurs without the payment completing successfully, then the monitor can automatically attempt to abort the payment.

3.4 Additional Payment Functions

From the point of view of the application, a payment is initiated by calling on method `StartTransfer` of the appropriate `PCR`, say for instance, `p`. This is natural since the `PCR` is the locus of control for the payment. However, it is difficult for a method in a generic payment record to actually execute the transaction, since it depends on the specific account types involved. We solve this problem by having `p.StartTransfer()` call a method `ah.StartTransfer(p)`, where `ah` is the source `AccountHandle`, i.e., where `ah = p.GetSourceAccountHandle()`. This latter method then actually makes the necessary calls to the underlying payment mechanism(s).

In summary, the following two methods of an `AccountHandle` can be invoked by the system, but should not be by the application programmer:

- `StartTransfer(PCR p)`
  This method is invoked by the `PCR`. All transfers should be initiated through the `StartTransfer` method of the `PCR`.

- `TryToAbortTransfer(PCR p)`
  This method is invoked by the `PCR`. All transfers should be aborted through the `TryToAbortTransfer` method of the `PCR`.

4 Sample Transaction

In this section, we will walk through the steps required for a typical transaction. These involve:

1. Creating an `AccountHandle` (done once)
2. Creating a `Monitor` object (done once or once per transaction)
3. Creating the `PCR` (Payment Control Record)
4. Initiating the transfer at the `PCR`
5. Initiating the transfer at the `AccountHandle`
6. Updating the status at the `PCR`
7. Calling back to the `Monitor` object

This transaction will be a typical "mail-order" one, with the merchant dictating the terms of the purchase, the customer placing an order and sending payment, and finally, the merchant sending the goods.

4.1 Creating an `AccountHandle`

In this example, the customer wishes to enable his First Virtual account to make payments within this system. His FV account has the account identifier "jsmith". There is a subclass of `AccountHandle`, called `FVAccountHandle`, for creating First Virtual accounts (developed by the people at First Virtual or a third party proxy-generator).

- `FVAccountHandle jmsFVaccthandle;`
  `(* This creates a new object *)`

- `FVAuth = jmsFVaccthandle.OpenAccount({"type: FV", "user-id: jsmith", "e-mail: jsmith@nowhere.net"})`

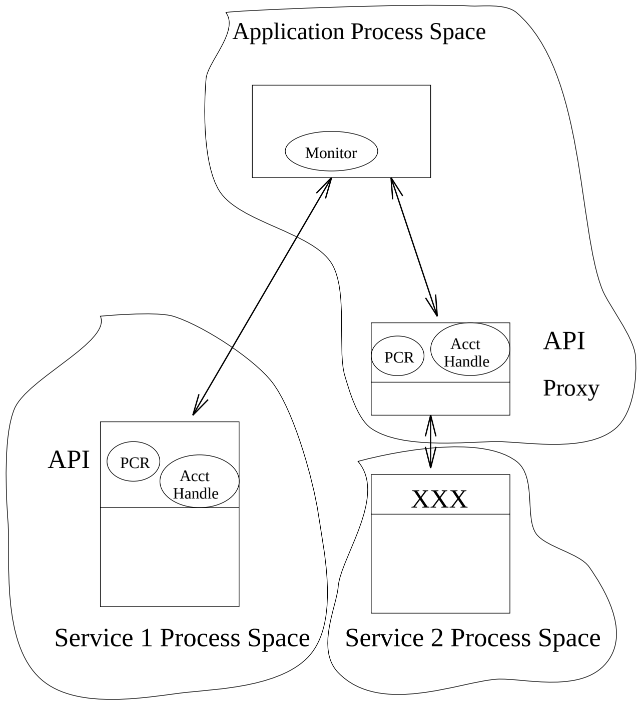

Figure 3: Location of system components in the system.

This `jmsFVaccthandle` object is a representation of the First Virtual account in the payment system. Its type is `FVAccountHandle`. The method implementations were written by the First Virtual development staff or proxy writers, but this object is now customized with J. Smith's account information. Further messages to it will result in communication with the First Virtual system to perform the desired operation and may rely on the return value of the `OpenAccount` method to provide authentication. The `AccountHandle` is located on the payment service side, or at the proxy, which may be running locally on the customer's machine. Figure 3 shows the location of key components in the distributed system.

4.2 Create a `Monitor` object

The buyer needs to have some method of keeping track of the status of various transactions. The `Monitor` object performs this role, located on the customer's machine, receiving updates as to the transaction status, and triggering application actions accordingly. For instance, if the payment is complete, the `Monitor` object should set up a process to receive the goods or complain if they are not received in a timely fashion. If payment stalls due to a problem such as insufficient funds, the `Monitor` object should choose an alternative payment mechanism if it is so authorized, or alert the application program to the problem.

In this example, we assume that the buyer will create a Monitor object exclusively for this transaction. The application programmer has developed a CustomerMonitor class which inherits from the Monitor object of U-PAI. The CustomerMonitor must support the one method of a Monitor object, Notify. The implementation details are application specific. A small piece of a typical monitor's Notify method is given here:

```
Notify(PCR whom, StatusEntry s):

 if s.MajorStatus == PaymentComplete:
    DeliveryMon.expect(whom.GetInvoice())
 elseif (s.MajorStatus == Failed) &&
   s.MinorStatus == "NotSufficientFunds"):
    self.SelectNewPaymentMech(whom)
```

The StatusEntry record has a MajorStatus field of enumerated type (PaymentComplete, InProgress, or Failed) and a MinorStatus which provides additional detail about the update. The Monitor object is responsible for determining what to do based on this new status. In the fragment above, it calls the application specific DeliveryMon if payment is complete, or tries to select a new payment mechanism if this one failed due to insufficient funds. These routines are both outside the scope of U-PAI.

The new CustomerMonitor object (fulfilling the role of the Monitor object) is created in the declarations before the transfer is started.

- CustomerMonitor CM;

The Monitor object should, at a minimum, support actions for each of the three MajorStatus values. If the application programmer knows in advance about specific payment mechanisms that will be used and the status values that they report (through the MinorStatus descriptions), the application can use this information in determining what step to take next.

4.3 Creating the PCR (Payment Control Record)

By creating a PCR, the application tells the payment component how much money should be sent to whom, from which account, and how to inform the application and other interested parties of updates. In this example, the customer has obtained the merchant's FVAccountHandle, authorization to deposit into that account, and a Monitor object (probably from an invoice or advertisement provided by the merchant) and has stored them in application variables MerchantAcctHandle, MerchantAuth and MerchantMonitors respectively. The amount that the customer intends to pay is \$4.00. The authorization code to use this source FVAccountHandle was generated by the OpenAccount method from Section 4.1 and was stored in FVAuth. Neither customer nor merchant needs to create a receipt for this transaction. The reference number is XE-2909, and is a payment for invoice number AXP-309. Due to the close relation between the PCR and the payment, the PCR is also typically located at the server or proxy.

- PCR pay;
  (*Creates the pay object of PCR type*)
- pay.SetDestAccountHandle (MerchantAcctHandle)
- pay.SetDestAuthorization(MerchantAuth)
- pay.SetSourceAccountHandle (jmsFVaccthandle)
- pay.SetSourceAuthorization(FVAuth)
- pay.SetMonitorList(MerchantMonitors $\cup$ {CM})
- pay.SetReceipts([])
- pay.SetRefID("XE-2909")
- pay.SetContextID("AXP-309")
- pay.SetAmount(4.00, "USD")

4.4 Initiating the transfer at the PCR

Once the buyer has completed the PCR object, he is ready to make a payment, and only one command is necessary.

- pay.StartTransfer()

If the buyer has filled in the SourceAuthorization field, anyone, including the merchant, can invoke the StartTransfer method. Once the PCR receives a request to initiate the payment, it passes it through to the AccountHandle, which has the appropriate payment mechanism-specific code to continue the operation.

4.5 Initiating the transfer at the `AccountHandle`

The `PCR` notifies the `Monitor` objects that the payment has been initiated, and they should expect additional updates on it. Then it interacts with the payment mechanism to accomplish the funds transfer. In this case, the `AccountHandle`, acting as a proxy, mimics the First Virtual protocol, generating, sending, receiving and processing e-mail messages (details in Section 5). Status updates will be sent to each `Monitor` object mentioned in the PCR's `MonitorList`, namely the `CustomerMonitor` instance `CM` and each `Monitor` object that the merchant supplied.

- `for m in self.MonitorList: m.Notify(self, [InProgress, "Payment Initiated"])`
- `(self.getSourceAccountHandle()). StartTransfer(self);`

4.6 Updating the status of the `PCR`

As the payment mechanism progresses through the steps of its internal process, it may periodically issue status updates to the `PCR`. It does this by means of the `UpdateStatus` method, invoked on the `PCR`. The `PCR` passes its own identity to the `Monitor` objects so that they can distinguish among the multiple transactions they may be monitoring. The payment mechanism interacts with the `AccountHandle` to trigger the status updates in the rest of the payment system.

- `thisPCR.UpdateStatus([PaymentComplete, "termination normal"])`

The identifier `thisPCR` is set to the `PCR` that was passed to the `AccountHandle` in the `StartTransfer` call. A payment mechanism may issue as many `InProgress` updates as it wishes, each with a different `MinorStatus` value. The payment mechanism or its proxy must make the `UpdateStatus` call when the payment terminates, either successfully or unsuccessfully.

4.7 Calling Back to the `Monitor` Objects

When the `PCR` receives a status update, it is responsible for echoing that update to each `Monitor` in its `MonitorList` field. When the transaction is successfully completed, the PCR's `Receipts` field is broadcasted to the monitors.

- `for m in self.MonitorList: m.Notify(self, [PaymentComplete, self.Receipts])`

The application may need to map back to the payment details of this transfer by getting its associated context object (such as the invoice), by invoking the `GetContextID` method on the `PCR`. When the `Monitor` object learns that the payment has completed, it takes the application specific behavior dictated in the `Notify` method. In our example, that involves calling the `DeliveryMonitor` to await the arrival of the ordered goods. The merchant's `Monitor` object would initiate the delivery of the order.

The transaction continues with the payment mechanism possibly making several `UpdateStatus` calls which are re-broadcast as `Notify` to the monitor list, until eventually the transaction completes with either a `Failed` or `PaymentComplete` status. At that time, the `PCR` is still accessible to the application, if it wishes to `GetStatus`, or the application may deallocate the space (garbage collect) the `PCR`.

5 Sample First Virtual Proxy

In this section, we show how one real world payment system can support this API without changing its current operation. The First Virtual (FV) payment mechanism (see `http://www.fv.com/`) was the first service which allowed consumers to transfer real money across the network, requiring both payer and payee to hold FV accounts. It works by assigning each user a new account name, and obtaining the user's credit card information in a secure, out-of-band channel. Designed primarily for information goods that merchants can produce and distribute for effectively zero marginal cost, the FV management encourages its merchants to give consumers a chance to "try before you buy", with the opportunity to refuse payment for the goods.

The full structure of a FV transaction (see Figure 4) is:

1. The customer sends his FV account information to the seller via e-mail.
2. The seller can optionally verify the existence of the account with FV, again by e-mail (optional, not shown in figure).
3. The seller delivers the goods to the buyer's e-mail address (which should match that of the FV account). This step is outside the scope of the payment process, and not shown in the figure.

4. The seller sends a charge request (via e-mail or telnet) to FV asking FV to bill the buyer.

5. FV sends an invoice to the holder of the FV account via e-mail.

6. The buyer responds by e-mail either indicating that he accepts the charge, acknowledges requesting the merchandise but does not want to pay for it, or does not recognize the charge and suspects fraud.

7. FV updates account balances if payment was approved, and informs the merchant of the resolution, using e-mail.

Some time later, FV aggregates the charges made by the user into a single charge to be levied on the associated credit card and paid to FV. Some time much later (90 days), the money is deposited in the appropriate merchant's checking account.

Figure 5 shows how the FV payment mechanism could interact with U-PAI. Steps labeled "A", "B", "C", and "D" correspond to the steps described in Sections 4.4, 4.5, 4.6, and 4.7 respectively. We assume for the sake of this example that the creation of the `AccountHandle` and the `Monitor` have been completed already. The application begins by creating a `PCR` (producing the object labeled as such in the figure) and then initiating a fund transfer (Step A), on the new `PCR`. In Step B, the `PCR` re-directs the call to the `AccountHandle` which acts as a proxy for First Virtual, receiving U-PAI messages, then translating them into the e-mail forms which are required by the First Virtual process. Forming another part of the FV proxy, the merchant's `AccountHandle` intercepts this mail message (Fig. 5, Step 1) and forms a FV invoice, which is sent to the FV commerce server (Step 4), possibly issuing a status update to the `PCR` as well. The FV service, ignorant that the e-mail invoice was automatically generated by the proxy, proceeds as it would if the invoice had come from a human, sending a copy of the invoice on to the specified customer, asking for approval (Step 5). Here part of the FV proxy working on the payer's machine intercepts the mail message and (assuming no `TryToAbortTransfer` invocation has been made) sends its approval to the FV server (Step 6), again with a possible update to the `PCR`. The FV server once again completes the processing, actually transfers the money, and sends the merchant e-mail describing the resolution. Here again, the merchant-side piece of the FV proxy intercepts the mail (Step 7), and must in this case `UpdateStatus` on the `PCR` (Step C) with the final resolution of the transaction, either `PaymentComplete` or `Failed`. After each status update at the `PCR`, the new information is passed to the `Monitor` objects (Step D) which take application specific behavior, possibly ignoring the `InProgress` updates, or informing the user. If the payment status is complete, then the PCR sends the information held in its `Receipts` field.

It is important to note that everything above the dotted line in Figure 5 is independent of the particular payment mechanism. In Figure 4, the application needed to know how to form e-mail messages to First Virtual. With the abstraction of an `AccountHandle` and `PCR`, however, (as we will see in the next section) a different payment mechanism could be substituted below the dotted line with no disruption to the application. This flexibility is the goal of U-PAI.

6 Sample Ecash Proxy

Developed by David Chaum of DigiCash, ecash is an electronic "coin"-based payment mechanism which provides anonymity for the purchaser. Although the technical details are complex [4], they are not directly of concern to U-PAI, which interacts with ecash at the level of the user operations. For this discussion, we assume the text-based interface to the system used in the cyberbucks ecash trial. The steps in an ecash payment are enumerated below, and shown graphically in Figure 6.

1. The payer initiates the payment by entering a command either in the ecash process or directly at the UNIX shell. The command specifies an amount, a destination host and port, and a reference string.

2. The ecash software withdraws an appropriate number of coins from the user's account to make the payment, and transmits them to an approved bank for verification.

3. Assuming the coins are legitimate and have not been spent, the payee (merchant) is asked to approve the the deposit.

4. The payee approves the deposit, sending a message to the ecash bank.

5. The bank sends the coins to the merchant, for deposit into the merchant's core account.

6. The payee application queries the payee account to determine whether the coins have arrived, using an ecash command entered directly from the UNIX shell or via the ecash interface.

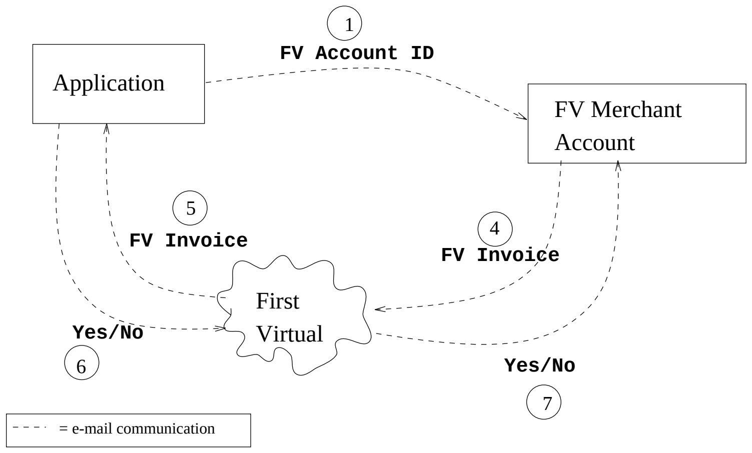

Figure 4: The steps involved in a First Virtual payment.

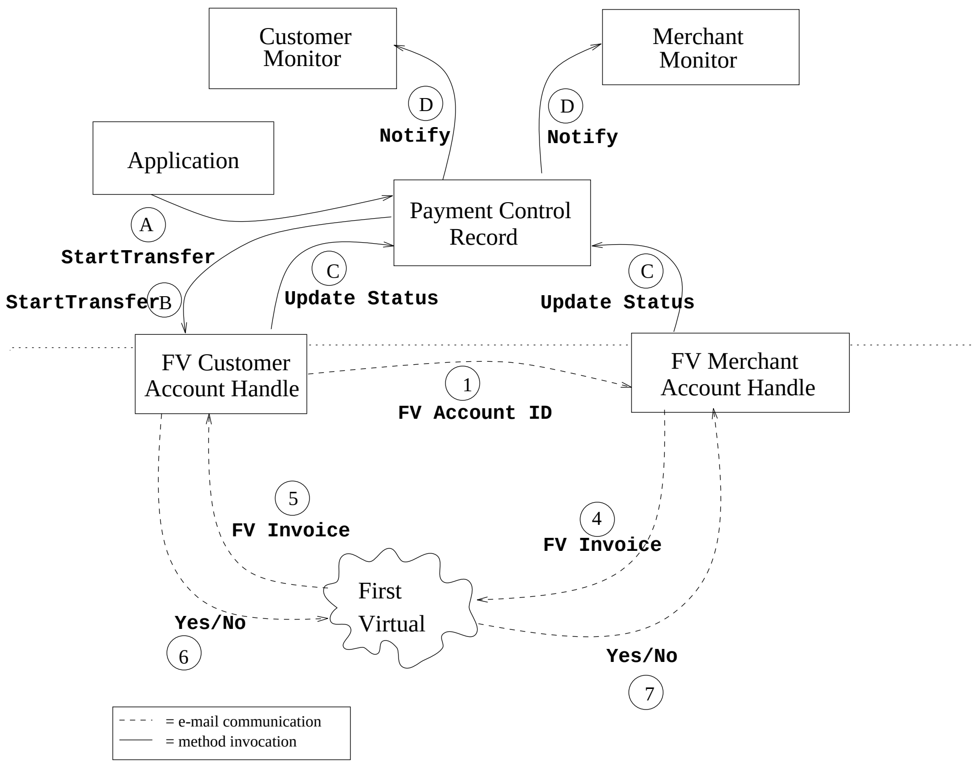

Figure 5: The steps involved in a First Virtual payment used with U-PAI.

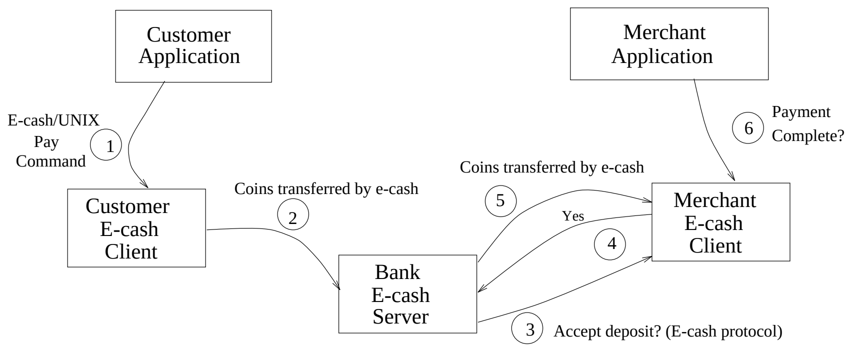

Figure 6: The steps involved in an ecash payment.

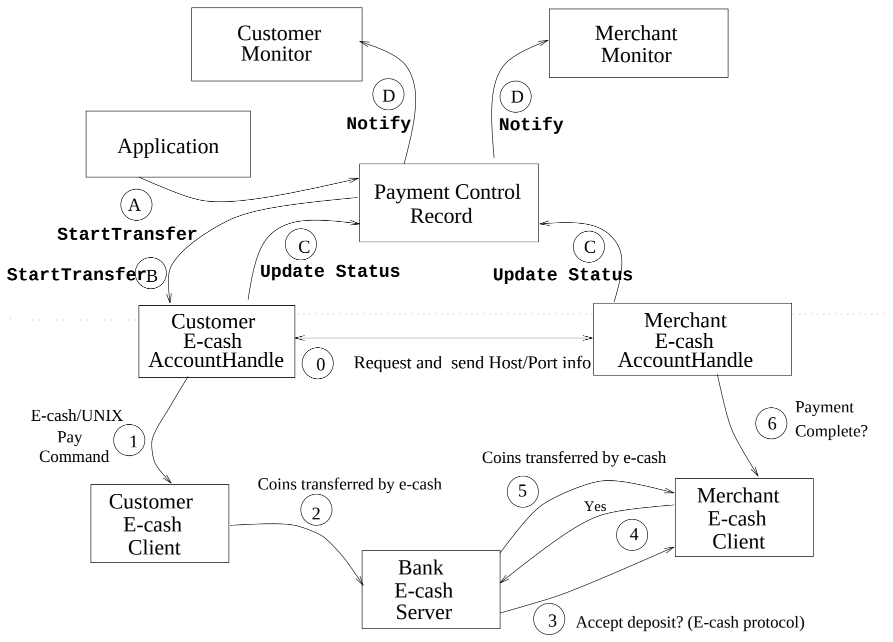

Figure 7: The steps involved in an ecash payment used with U-PAI.

The AccountHandles act as the proxy to the payment mechanism, as with the First Virtual system. When the StartTransfer is invoked, the destination AccountHandle is available to the source AccountHandle. The source AccountHandle calls a mechanism-specific method (not part of the definition of U-PAI) defined on ecash AccountHandles to learn the host address and port of the destination account (Fig. 7, Step 0). With this information, the source AccountHandle formats and executes an ecash pay command (Step 1). At this point, the ecash module takes over, contacts the bank, and verifies the coins (Step 2). The request for approval which the ecash bank sends to the payee (Step 3) is intercepted by the proxy on the merchant side, and automatically approved in Step 4 (if automatic approval is not acceptable on all payments, the source AccountHandle can notify the destination AccountHandle of the coming payment). The coins are then transferred to the merchant (Step 5), again through ecash specific code. The merchant's ecash AccountHandle determines when the payment is complete (Step 6) and triggers an update to the PCR (Step C). The status update from the PCR is sent to the monitors on the MonitorList (Step D), allowing the applications to be informed of the final disposition of the ecash payment, using the same status values (PaymentComplete, Failed, or InProgress) from the First Virtual proxies. If the status is PaymentComplete, then the PCR distributes the information recorded in its Receipts field. The interaction of the ecash system with the U-PAI interface is shown in Figure 7. Again, notice that the machinery above the dotted line is identical to that in Figure 5.

7 Failed Transactions and Security

In this section, we consider the behavior of the system in a few selected failure modes, such as network disturbances or frozen accounts. In some cases, the system design allows completion of a commercial transaction even under adverse circumstances. For instance, in the event that the ecash bank server is down, the StartTransfer method will recognize its inability to contact the bank, and notify the listening Monitor objects, perhaps enabling the buyer to select a different payment mechanism which is currently operable. Similarly, an ecash charge for which there are insufficient funds will result in an error condition being sent to the Monitor object, enabling alternative arrangements to be made.

The system is not foolproof, however. If a user initiates a payment using First Virtual and then receives no update because the e-mail was delayed, the user is uncertain of what to do. The status may show only InProgress with no indication of what step is currently ongoing, or how much longer is required before the process will be resolved. This ambiguity highlights one of the design decisions of U-PAI. In an effort to promote ease of implementation, no guarantees are offered about the completion of transactions—the mechanism and system operate on the "best effort" principal. In particular, under certain failure conditions with certain payment mechanisms, it may be impossible for the payer to prove that the payee received payment. By providing fine-grained specification of the transaction status to the Monitor objects through the MinorStatus values, however, along with the power to abort a transfer, the system provides maximum flexibility to its users. If a payment mechanism provides the capability to query the status of a particular transaction, an additional level of recovery is possible, because a Monitor object can use the AccountHandle's GetStatus method if the PCR fails.

Also, security is not explicitly discussed in this paper. For the CORBA-based U-PAI methods (above the dotted line in Figs. 5 and 7), we assume the presence of a mechanism which provides access control on a per-method, per-object basis. This may be implemented using access capabilities building on the Authorization fields of the PCR. Other mechanisms such as digital signatures may be substituted. The desired result is that certain objects are prevented from reading or modifying data fields or executing methods, while other objects are permitted partial or total access. For example, the UpdateStatus method on a PCR should only be called by AccountHandles involved in the transaction.

For those steps below the dotted line, we assume that the underlying payment mechanism handles security appropriately. Properties such as confidentiality and non-repudiation that are provided by the payment mechanism may require additional work to ensure they persist through U-PAI. The messages should also be encoded in such a way to resist eavesdroppers and replay attacks.

8 Conclusion

We have proposed a Universal Payment Application Interface, which allows a variety of payment mechanisms to be accessed by the same interface, easing the use of multiple payment mechanisms or the process of switching between payment mechanisms. We have outlined how payment mechanism proxies (combination of modules on both user and merchant side) allow this API to be supported without modification of the underlying payment mechanism. Finally, we have provided a CORBA ISL file for programmers interested in supporting or using this interface.

9 Acknowledgments

The authors wish to thank Ali Bahreman for thought-provoking discussions and helpful suggestions. Also, Martin Röscheisen contributed to the design of InterPay[3], a preliminary version of this work. Finally, changes suggested by the referees improved the completeness and clarity of this work.

References

[1] Alireza Bahreman. Generic payment services: Framework and functional specification. Technical report, 1996. Proceedings of the Second USENIX Electronic Commerce Workshop.

[2] Alireza Bahreman and Rajkumar Narayanaswamy. Payment method negotiation service: Framework and programming interface. Technical report, 1996. Proceedings of the Second USENIX Electronic Commerce Workshop.

[3] 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.

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

[5] Donald E. Eastlake, 3rd. Universal payment preamble. Technical report, CyberCash, Mar 1996.

[6] Steven H. Low, Nicholas F. Maxemchuk, and Sanjoy Paul. Anonymous credit cards. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, 1994.

[7] Mark S. Manasse. The Millicent protocols for electronic commerce. In *Proceedings of the 1st USENIX workshop on Electronic Commerce*, 1995.

[8] Andreas Paepcke, Steve B. Cousins, Hector Garcia-Molina, Scott W. Hassan, Steven P. Ketchpel, Martin Röscheisen, and Terry Winograd. Towards interoperability in digital libraries: Overview and selected highlights of the Stanford digital library project. *IEEE Computer Magazine*, 29(5), May 1996.

[9] 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.

10 Appendix: CORBA Payment Mechanism ISL

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

TYPE String = ilu.CString;

TYPE Amount = RECORD
  Number : REAL,
  Units : String  (* dollars, yen, etc *)
END;

TYPE RefIDType = String;

TYPE AccountTypeID = String;

TYPE AccountTypeIDList = SEQUENCE OF AccountTypeID;

TYPE Monitor = OBJECT
  METHODS

    Notify(whom : PCR, status : StatusEntry)
      (* Notify is called whenever the status of the transaction 'whom'
         changes & this Monitor object was in the PCR. *)
  END;

TYPE MonitorList = SEQUENCE OF Monitor;

TYPE AccountHandle = OBJECT
  METHODS

    CreateAccount(NewAccountInfo : CosPropertyService.PropertySet): IAny.Any,
      (* Creates a new real-world account, with the appropriate identifying
         information.  Optionally returns an authentication token.*)

    OpenAccount(AccountInfo : CosPropertyService.PropertySet): IAny.Any,
      (* Creates a new electronic representation of the existing real-world
         account with the appropriate identifying information.
 Optionally returns an authentication token.*)

    GetAccountType() : AccountTypeID,
      (* returns the type of this account. *)

    GetTransferAccountTypesFrom() : AccountTypeIDList,
      (* returns a list of account types that this account can receive
         money from. *)

    GetTransferAccountTypesTo() : AccountTypeIDList,
      (* returns a list of account types that this account can transfer to. *)

    GetBalance() : Amount,
      (* returns the amount of funds available for payment in this account. *)

    GetCreditLimit() : Amount,
      (* returns the credit limit for credit-based accounts. *)

    GetMechanismProperties() : CosPropertyService.PropertySet,
      (* returns the meta-data properties like cost, time, anonymity. *)

    CloseAccount(),
      (* close this account.  No further transfers can be made. *)

    DeleteAccount(),
      (* close this account & eliminate the real world account, too*)

    StartTransfer(p : PCR),
      (*  Called by the system, not application programmer, to start
          the money transfer *)

    TryToAbortTransfer(p : PCR),
      (*  Called by the system, not application programmer, to try to abort
          the money transfer *)

    GetStatus(RefID : RefIDType) : PaymentStatus
      (* Returns the current status of the payment identified by RefID. *)


  END;

TYPE MajorType = ENUMERATION
         PaymentComplete,          (* Money transferred from payer to payee*)
         InProgress,               (* Transfer started, not completed *)
         Failed                    (* Error in payment, see description field*)
    END;

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

  (*Typical Values are strings:
         Aborted                     -- Payer requested abort
         NotSufficientFunds,         -- Not Sufficient Funds for payer
         UnauthorizedSourceAccount,  -- Payer not authorized to make payments
                                        from this account
         UnauthorizedDestAccount,    -- Payer not authorized to make deposits
                                        to this account
         NonExistentDestinationAccount -- Payee account not recognized
         UnableToTransferToAccountType -- Payee account wrong type
         NoSourceAccountSelected     -- Neither open () nor create()
                                        has been invoked on this handle
   *)

    END;

TYPE PaymentStatus = SEQUENCE OF StatusEntry;

TYPE PCR = OBJECT

  METHODS

    SetRefID(RefID : RefIDType),
    SetContextID(ConID : RefIDType),
    SetAmount(amt : Amount),
    SetMonitorList(Mlist : MonitorList),
    SetDestAccountHandle(dest : AccountHandle),
    SetDestAccountAuthorization(auth : IAny.Any),
    SetSourceAccountHandle(src : AccountHandle),
    SetSourceAccountAuthorization(auth : IAny.Any),
    SetReceipts(rcptlist : IAny.Any),

    GetRefID() : RefIDType,
    GetContextID() : RefIDType,
    GetAmount():Amount,
    GetMonitorList():MonitorList,
    GetDestAccountHandle():AccountHandle,
  GetDestAccountAuthorization() : IAny.Any,
  GetSourceAccountHandle() : AccountHandle,
  GetSourceAccountAuthorization() : IAny.Any,
  GetReceipts() : IAny.Any,

  StartTransfer(),
    (* Initiates the transfer described in the other fields of the
       data structure.  Asynchronous, returning immediately, doesn't wait
       for funds to be transferred. *)

  GetStatus() : PaymentStatus,
    (* Returns the current status of this transaction. *)

  TryToAbortTransfer(),
    (* Attempts to abort the transfer of funds initiated
       for this PCR.  There is no guarantee the abort
       will be successful.  *)

  UpdateStatus(stat : StatusEntry)
    (* Called by payment specific level to report progress *)
END;
```