Modeling Exceptional Behavior

Neelam Soundarajan and Stephen Fridella

Computer and Information Science
The Ohio State University
Columbus, OH 43210
e-mail: {neelam,fridella}@cis.ohio-state.edu

Abstract. While exception handling mechanisms are very useful for implementing systems, they are equally useful when building business models. The standard method of modeling exceptions in UML is to include them in class diagrams as a kind of *signal*, as stereotyped classes. However this alone is insufficient, since by the very nature of exceptions, the circumstances under which particular exceptions may be raised, as well as the details of what actions the class handling the exception will perform in doing so, tend to be rather complex and in general difficult to express pictorially. In this paper, we consider how this information may be specified using the *Object Constraint Language (OCL)*. We illustrate our approach by applying it to a simple example.

1 Introduction

Exception handling mechanisms are useful for *implementing* systems, and indeed that is why several OO languages such as *Java, C++*, and *Eiffel* provide such mechanisms. Exceptions are useful also when building business models, in other words at, what Fowler [3] calls, the *conceptual* level. At the implementation level exceptions are useful because they allow us to structure systems so that the normal code is cleanly separated from the exception handling code, thereby making the system easier to understand and to maintain. It is for exactly the same reason that exceptions are useful at the conceptual level too; the OO designer can first design the business model taking account of only the normal cases; and once this is done, he[1] can extend the model, by adding appropriate exceptions, to deal with the exceptional cases.

How do we include information about these exceptions in the UML diagram for the application? The standard approach [1] is to model exceptions as a special kind of *signal*, as stereotyped classes. But exceptions, whether in the implementation or in the business model, can be difficult to understand. For this reason, representing exceptions as signals in UML diagrams, although it conveys much useful intuition about which classes may raise which exceptions, by itself is not

[1] Following standard practice, we use he, his etc. as abbreviations for 'he or she', 'his or hers' etc.

always sufficient. It is important to characterize more precisely, the conditions under which specific exceptions may be raised, as well as what information will be available when particular exceptions are raised. In this paper we investigate ways of doing this using the *Object Constraint Language (OCL)* [5].

A standard OCL specification of a class C consists of precise specifications in the form of *pre-* and *post-* conditions for each of the operations that C provides, and that may be used by its clients. The pre-condition of each operation specifies the condition that must be satisfied at the time that the operation is invoked; this condition may involve the values of any arguments that the operation receives, as well as the values of any of the attributes of C. The post-condition of the operation similarly tells us what conditions will be satisfied by the values of the attributes of C when the operation finishes. And, in the case of operations that return a result, it also provides information about the result. The pre- and post-condition specifications embody the principle of *Design by Contract (DBC)* [4]: If the client (the caller) satisfies his part of the contract, that of ensuring the pre-condition of the operation at the time of the call to the operation, then (and only then) the service provider guarantees his part of the contract, that of ensuring the post-condition at the time the operation finishes. *Nothing* is guaranteed if at the time of the call to the operation, its pre-condition is not satisfied.

What do we do if this operation raises an exception? One approach would be to think of exceptions as 'catastrophic' events. In other words, an exception essentially tells the client class that the invoked operation *failed* for some reason. In effect this is what treating an exception as a particular kind of signal without providing any additional information amounts to; indeed, the icon that the three amigos suggest [1] for these signals is the classic 'danger' icon, consisting of a big exclamation point inside a triangle. Again, in *Eiffel* [4], exceptions are treated as failures (although by using some standard library facilities it is possible for the operation of an *Eiffel* class to provide some modest indication to the client, of the reason for its failure). While exceptions corresponding to such events as, say, power failures during the execution of the actual implementation model may qualify as catastrophes, languages like *Java* and *C++* allow for a more flexible treatment of exceptions. In these languages, exceptions are full-fledged objects, and can be used in very flexible ways to pass information back to the client including, in particular, information about the reason for the exception being raised. Similarly, considering exceptions as full-fledged objects (rather than merely signals with little information content), allows us to exploit them in interesting ways in the business model. In the next section we will see a simple banking example in which we use exception objects in this manner.

We also need to see how to specify exactly the conditions under which the exceptions will be raised, as well as precisely what information the exception object will contain if it is raised. In section 3 we will propose a simple and natural extension to OCL specifications to include such information. In section 4 we will return to the banking example and see how it can be specified using our notation, and how the information contained in the specification can be used by the designer of the client class. In the final section we reiterate the importance of exceptions as full-fledged objects in OO design, summarize our approach to extending OCL specifications to include information about operations that raise exceptions, and briefly consider some other aspects of UML that need to be enriched to deal with exceptions.

2 Banking Example

Consider the simple banking example shown in Figure 1. As the figure shows, a Bank has one or more Accounts; each Account is held by a single Customer (in other words, no joint accounts); and a Customer has zero or one Account. The Account class provides standard operations for checking the current balance in the account, and for depositing and withdrawing money from the account. The Account class is simple enough that one might be tempted not to include any further information about the model than is provided in Figure 1. But even such a simple example presents potential problems. For example, does the bank impose any service charges on the transactions? If yes, do the charges apply to both withdrawals and deposits? Do the charges depend in any way on the balance in the account?

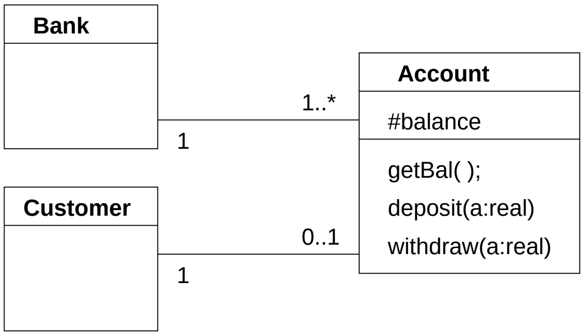

Fig. 1. Bank and Account classes

Suppose that the bank does impose a service charge of, say, 1.00 (one dollar) on each transaction but only if the current balance in the account is below 1000.00. In Figure 2, we use the OCL notation to specify this behavior in the form of *pre-* and *post*-conditions of the particular operations. The pre-condition of each operation specifies the condition that must be satisfied at the time that the operation is invoked; this condition may involve the values of any arguments that the operation receives, as well as the the values of any attributes of the Account class. Thus the pre-condition of the withdraw(a) operation, for example, tells us that we can only withdraw positive amounts; this pre-condition does not

$$\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{getBal}() : \mathsf{real}} \\
\mathsf{pre} : \mathsf{true} \\
\mathsf{post} : (\mathsf{result} = \mathsf{balance}) \wedge (\mathsf{balance} = \mathsf{balance@pre})
\end{array}$$

$$\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{deposit}(\mathsf{a{:}real}) : \mathsf{void}} \\
\mathsf{pre} : \mathsf{a} > 0.0 \\
\mathsf{post} : (\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} + \mathsf{a}) \\
\qquad\quad \wedge (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} + \mathsf{a} - 1.0)
\end{array}$$

$$\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{withdraw}(\mathsf{a{:}real}) : \mathsf{void}} \\
\mathsf{pre} : \mathsf{a} > 0.0 \\
\mathsf{post} : (\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \\
\qquad\quad \wedge (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a} - 1.0)
\end{array}$$

Fig. 2. Specification of Account class

impose any requirements on the values of the attributes of the account at the time the operation is applied, but we will change that shortly.

The post-condition of the operation similarly tells us what conditions will be satisfied by the values of the attributes of Account when the operation finishes. And, in the case of operations that return a result, it also provides information about the result. OCL provides two important notations for use in post-conditions. First, the keyword result is used for referring to the result returned by an operation. Second, since the values of the attributes of the class when the operation finishes, are usually related to their values at the start of the operation, OCL provides the @pre notation in terms of which such relations can be easily expressed: in OCL specifications, x@pre in the post-condition denotes the value of x at the time the operation started execution, whereas x in the post-condition denotes the value of x at the time the operation finishes. Thus the post-condition of getBal() tells us that the result returned by the operation is the same as balance, and that the final value of balance is the same as when the operation started. Similarly, the post-condition of withdraw(a) tells us that the balance in the account is appropriately updated, i.e., it is reduced by either a or by $(\mathsf{a} + 1.0)$ depending on whether the balance in the account at the time the operation was invoked (i.e., balance@pre) was not or was less than 1000.0.[2]

In practice, it is frequently the case that the value of one or more attributes of the class when an operation finishes, are the same as at the start of the operation. To make it easy to specify such behavior, we use one other notation

[2] OCL allows us to write the two clauses of the post-condition as two separate post-conditions for the operation, one corresponding to the case when balance@pre was less than 1000.0, the other corresponding to when it was not. As a matter of style, we prefer to combine all the cases into a single post-condition to ensure that we don't overlook any particular case. But this is purely a matter of taste. Similarly when writing what we will call exception-post-conditions, we will combine all the cases into a single post-condition, although it would be perfectly possible to write them as distinct post-conditions.

in post-conditions, although it is not part of OCL: in post-conditions, $!\{\mathsf{x}\}$ will mean that the value of x when the operation finishes is the same as when it started; thus this is equivalent to $(\mathsf{x} = \mathsf{x@pre})$. Similarly, $!\{\mathsf{x,y}\}$ means that the final values of x and y are the same as their corresponding initial values; in strict OCL this would be written as: $(\mathsf{x} = \mathsf{x@pre}) \wedge (\mathsf{y} = \mathsf{y@pre})$ Using the '!' notation, the post-condition of getBal() could have been written as:

$(\mathsf{result} = \mathsf{balance}) \wedge !\{\mathsf{balance}\}$

Consider again the withdraw(a) function. What happens if it is invoked with an argument value that is greater than the current balance? According to the specification in Figure 2, such an operation is indeed allowed and the result will be that the value of balance will be reduced by the amount a (or $(\mathsf{a} + 1.0)$, depending on balance@pre) and hence will become negative. Real banks don't, of course, allow all their customers arbitrary overdrafts, so let us change the model to make it a bit more realistic. One simple approach would be make the pre-condition of withdraw() more demanding:

$$\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{withdraw(a{:}real)} : \mathsf{void}} \\
\mathsf{pre} : (\mathsf{a} > 0.0) \wedge (\mathsf{balance} \geq \mathsf{a} + 1.0) \\
\mathsf{post} : (\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \\
\qquad\quad \wedge (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a} - 1.0)
\end{array}$$

Given such a specification, we do not, in the withdraw(a) operation, have to worry about the possibility that there might be insufficient funds in the account. While this approach is not incorrect, it is rather inflexible. Rather than strengthening the pre-condition of Account::withdraw(), a more flexible design, would be for the Account class operation to raise an *exception* if there are insufficient funds in the account, and let the client class, which in this example will be the Bank class, decide how proceed in this case.

But what does it mean to raise an exception? What can the client class do if an exception is raised? Suppose, for example, that the Bank class contains an operation that is carrying out a whole series of transactions on different Account objects by invoking deposit() and withdraw() operations on the corresponding Account objects. And suppose one of these withdraw() operations raises an exception because the particular account has insufficient funds. If the exception were just a signal of some kind indicating to the Bank class that one of the transactions failed, that does not allow the Bank class to take appropriate action; indeed, since we would expect the exception-handling part of the Bank class operation in question to be separate from the main body of that operation, this part may not even know which account object had insufficient funds that resulted in the exception being raised. Hence, a more useful design would be for the exception to include additional information such as, for example, a reference to the particular account that was the source of the problem; that would allow the exception handling portion of the Bank operation to take suitable corrective action – such as freezing the account in question, or allowing the overdraft if the past history of the account warrants it, or simply not performing that particular transaction but continuing with the rest, etc. Of course some of these courses of action (such as freezing the account) will require the Account class to provide suitable operations and we will assume that such operations have been added to the Account class.

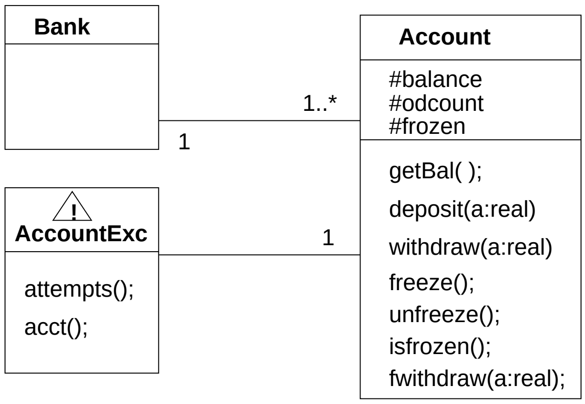

Fig. 3. Bank, Account, and AccountExc classes

In Figure 3, we have introduced the exception class AccountExc (and, in the interest of reducing clutter, we have omitted the Customer class). We have also made some changes in the Account class; let us consider these changes first. We have added two attributes #odcount and #frozen to the Account class. These will keep track respectively of the number of attempts made to overdraw funds from this particular account, and whether this account is currently frozen or not. We have also introduced four new operations: freeze(), unfreeze(), isFrozen(), and fwithdraw(); the first allows the client (the Bank class) to freeze an account, the second unfreezes the account, the third tells us if the account is frozen, and the fourth allows a *forced* withdrawal, essentially an overdraft.

The AccountExc is a fairly simple class. First note that we have flagged this as an exception class by including the 'caution' icon. The class provides two operations; the first, attempts(), returns the number of attempts that have been made to overdraw funds from the account that caused this exception to be raised. The second, and more interesting is the operation acct() which returns (a reference to) the account object that caused the exception. In the figure we did not indicate the attributes in which this information is maintained; we will introduce them when we specify these classes more precisely.

If the withdraw() operation is invoked with the balance in the account being insufficient, or if the account is currently frozen, the operation will create a new instance, which we will often refer to as ε, of the AccountExc class and 'throw' it. This exception object ε will include a reference to this particular account, and it is this reference that the acct() operation will return when it is applied to the object ε. This will allow the exception handling code in the client class to know which particular account was responsible for the exception being raised, and can then proceed to freeze the account or perhaps, if the the number of overdraw attempts is not too high, allow an overdraft (using fwithdraw()), etc.

But how do we specify all this information? Since it is the withdraw() operation that creates and throws the exception, it would be reasonable to except the specification of that operation to provide this information. In the next section, we will extend the pre- post-condition OCL specification of operations to provide information about exceptions that the operation may raise. Essentially, we will allow the specification of an operation to include a set of *exception-* or *e-post-conditions*, one corresponding to each exception that the operation may raise. The e-post-conditions will give us information about the conditions that will be satisfied if the operation throws that particular exception. As we will see, this post-condition can also, by judiciously using the @pre mechanism, allows us to specify the conditions under which this exception will be thrown. In section 4 we will return to the banking example and precisely specify the behaviors of the classes of Figure 3.

3 Specifying Exceptions

Suppose C is a class and f is one of its operations. The standard OCL specification of f specifies its pre- and post-conditions. Such a specification is appropriate only if f is guaranteed to terminate normally, i.e., without raising an exception (provided, of course, that its pre-condition was satisfied when it began execution). Suppose instead that f may either terminate normally or it may terminate by raising an exception of type E_1, or of type E_2, or of type $\ldots$, E_n. We will specify such an f as follows:

$\underline{C :: f}$
pre : ...
post : ...
E_1.post : ...
E_2.post : ...
$\vdots$
E_n.post : ...

The normal post-condition (which we will refer to as the n-post-condition) post, will, as usual, be a condition on the values of the attributes of C at the time that f terminates normally and the result, if any, returned by f (as well as, by using the @pre notation, the values of the C's attributes at the time f started execution). The e-post-condition E_1.post will be a condition on the values of the attributes of C at the time that f terminates by creating a new exception ε of type E_1 and raising it, and the values of the attributes of ε at the same time; E_1.post may also refer, using the @pre notation, to the values of the attributes of C at the time that f started execution, but it may not refer to the result returned by the operation. This is because if f terminates by raising an exception, it does not return a result. It *does* return (or, rather, 'raises') the exception object ε, so we include information about the values of the attributes of this object in $E_1.\mathsf{post}$. The other e-post-conditions are similar.

This specification should be interpreted as follows: If the execution of f begins with the condition specified in pre being satisfied, then it is guaranteed that one of the following will happen: Either the execution of f will terminate normally with the values of the attributes of C at the time of termination (and the value returned by f if any) satisfying the condition specified in post; Or the execution of f terminates by creating and throwing an exception of type E_1 or $E_2, \ldots, E_n$ with the values of the attributes of C at the time of termination and the state of the exception object ε satisfying the condition specified in $E_1.\mathsf{post}$ or $E_2.\mathsf{post}$ or ... or $E_n.\mathsf{post}$, respectively.

A number of points are worth noting. First, at most one exception may be raised by an operation. Thus, either the operation terminates normally *or* it terminates by raising a single exception. There are no other possibilities.

Second, the raising of an exception does *not* mean that the design-by-contract principle is being violated in any sense. Indeed, the kind of specification we have introduced allows us to characterize the exact conditions under which various types of exceptions may be raised. Consider a simple (but completely artificial) example:

$$
\begin{array}{l}
\underline{C \ :: \ f : \mathsf{int}} \\
\mathsf{pre} \ : (0 \leq \mathsf{x} \leq 2) \\
\mathsf{post} \ : \ (\mathsf{x@pre} = 0) \land (\mathsf{x} = 1) \land (\mathsf{result} = 1) \\
E_1.\mathsf{post} \ : (\mathsf{x@pre} = 1) \land (\mathsf{x} = 2) \land (\varepsilon.\mathsf{y} = -1) \\
E_2.\mathsf{post} \ : (\mathsf{x@pre} = 2) \land (\mathsf{x} = 2) \land (\varepsilon.\mathsf{z} = -2)
\end{array}
$$

This specification tells us that $C.f$ maybe invoked only if the value of the x attribute of the C object is between 0 and 2. If this condition is satisfied when f is invoked, this specification assures us that this invocation of f may terminate normally, or it may terminate by raising an exception of type E_1, or by raising an exception of type E_2. f will terminate normally provided the value of the x attribute when f started execution was 0; and in this case, f will return 1 as the result, and the value of x when f finishes will be 1. If, on the hand, the value of x at the time that f started execution, was 1, then f will terminate by throwing an exception of type E_1, the value of x at that time will be 2, and the y attribute of the thrown exception object will be -1. And, finally, if the the value of x at the time that f started execution, was 2, then f will terminate by throwing an exception of type E_2, the value of x at that time will be 2, and the z attribute of the thrown exception object will be -2. Note also that nothing is asserted about what will happen if f is invoked with the pre-condition not being satisfied. That is outside the contract.

Third, in this particular example, the three possible outcomes are disjoint.

That is, for any particular value (within the range allowed by the pre-condition) for x when f started execution, exactly one outcome is possible. But this is a feature of this example; our specification notation does not require it. Suppose, for example, that we replaced the clause (x@pre = 2) in E_2.post by [(x@pre = 2) $\vee$ (x@pre = 1)], and made no other changes, that would mean that if the initial value of x was 1, the operation would terminate by raising an exception of type E_1 or of type E_2. It is important to be able to allow such freedom in the specification because this allows us to build flexible business models. But note that 'flexibility' does not mean 'imprecise'; the specification, with this substitution, is as precise as the original specification. It is just that it leaves the freedom, to the implementor of the class, of raising either the exception of type E_1 or of type E_2 if f were to start with x being 1. D'Souza [2] also points out the importance of such flexibility.

Clearly, our approach to specifying and modeling exceptions is quite different from that of treating exceptions as 'failures' of some kind. Meyer [4], for example, essentially treats an exception as a violation of the contract by the operation raising the exception. We believe that if we are to exploit the power of using exceptions as full-fledged objects that pass useful information back to the client class (beyond the minimal fact that the operation 'failed'), then rather than treating it as a violation of the contract, it is more useful to extend the contract to cover this case; in other words, precisely specify under what conditions which exceptions may be thrown, and precisely specify information about the state of the specific exception objects that may be thrown.

4 Specifying Account and AccountExc classes

In this section we will specify the behavior of the Account class and AccountExc class of Figure 3 in terms of n- and e-post-conditions. To make the presentation simple, we will omit the account-balance based transaction fee that we considered in our earlier discussion. Before looking at the formal specifications, let us consider the behavior of each of the Account class operations informally.[3]

The attribute balance represents the current balance in the account; odcount gives us the number of attempts made so far to overdraw funds from this account; the value of frozen indicates whether the account is currently frozen. The operation getBal() returns the current balance. deposit(a) adds the specified amount

[3] There is nothing wrong with doing this. Just because we have a notation for formal specifications does not mean we should not also work with informal explanations. Indeed, the whole idea is to use several different ways of looking at the the classes and the operations – the UML class diagrams, informal explanations of what the operations do, and the formal specifications. And other techniques such as use-case diagrams provide additional handles to help us understand the behavior of the entire system; indeed, that is the point of the *unified* modeling language. We will return to this point in the final section.

to the balance. The operations freeze() and unfreeze() allow the client to change the value of frozen. The operation isFrozen() returns the current value of that attribute. These operations are all specified in Figure 5.

$\underline{\text{Account :: getBal() : real}}$
pre : true
post : (result = balance) $\wedge$!{balance, odcount, frozen}

$\underline{\text{Account :: deposit(a) : void}}$
pre : (a > 0)
post : (balance = balance@pre + a) $\wedge$!{odcount, frozen}

$\underline{\text{Account :: isFrozen() : bool}}$
pre : true
post : (result = frozen) $\wedge$!{balance, odcount, frozen}

$\underline{\text{Account :: freeze() : void}}$
pre : true
post : (frozen = true) $\wedge$!{balance, odcount}

$\underline{\text{Account :: unfreeze() : void}}$
pre : true
post : (frozen = false) $\wedge$!{balance, odcount}

Fig. 4. Specification of Account class (part 1)

Note that in the n-post-condition of each operation we have used the '!' notation to specify which variables are unmodified from their original values. Note also that there are no e-post-conditions in any of these operations; that means that none of these operations may raise an exception (at least not if their respective pre-conditions are satisfied when the operations are invoked).

The interesting operations are of course, withdraw(a) and fwithdraw(a). Since these might throw exceptions of type AccountExc, let us first consider the attributes of that class. This class has two attributes, noattempts, and account. The second attribute contains a reference to the particular account object on which an operation was being attempted that resulted in this exception being thrown. The value of the first attribute is the number of attempts that had so far been made to overdraw funds from that particular account at the time that the exception was thrown.

The operation withdraw(a) works as follows: if the account is currently frozen, it raises an exception of type AccountExc; this exception object has its account attribute set equal to the account in question, and the noattempts attribute set equal to the current value of the odcount attribute of this account. If the account is not currently frozen, and if the current balance in the account is greater than or equal to the amount a, then balance is appropriately updated, and the operation returns normally. If the balance is insufficient, the value of odcount is incremented by 1, and an exception of type AccountExc is raised. The operation fwithdraw(a) is similar, except that it does not raise an exception if the balance is insufficient.

$$
\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{withdraw(a{:}real)} : \mathsf{void}} \\
\mathsf{pre} : \mathsf{a} > 0 \\
\mathsf{n\text{-}post} : (\mathsf{frozen@pre} = \mathsf{false}) \wedge (\mathsf{balance@pre} \geq \mathsf{a}) \\
\qquad \wedge (\mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \wedge !\{\mathsf{frozen}, \mathsf{odcount}\} \\
\mathsf{AccountExc.post} : \\
\qquad ((\mathsf{frozen@pre} = \mathsf{true}) \vee (\mathsf{balance@pre} < \mathsf{a})) \wedge !\{\mathsf{balance}, \mathsf{frozen}\} \\
\qquad \wedge ((\mathsf{frozen} = \mathsf{true}) \Rightarrow (!\{\mathsf{odcount}\}) \\
\qquad \wedge (\mathsf{frozen} = \mathsf{false}) \Rightarrow (\mathsf{odcount} = \mathsf{odcount@pre} + 1)) \\
\qquad \wedge ((\varepsilon.\mathsf{account} = \mathsf{self}) \wedge (\varepsilon.\mathsf{noattempts} = \mathsf{odcount}))
\end{array}
$$

$$
\begin{array}{l}
\underline{\mathsf{Account} :: \mathsf{fwithdraw(a{:}real)} : \mathsf{void}} \\
\mathsf{pre} : \mathsf{a} > 0 \\
\mathsf{n\text{-}post} : (\mathsf{frozen@pre} = \mathsf{false}) \wedge (\mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \wedge !\{\mathsf{frozen}, \mathsf{odcount}\} \\
\mathsf{AccountExc.post} : \\
\qquad (\mathsf{frozen@pre} = \mathsf{true}) \wedge !\{\mathsf{balance}, \mathsf{frozen}, \mathsf{odcount}\} \\
\qquad \wedge ((\varepsilon.\mathsf{account} = \mathsf{self}) \wedge (\varepsilon.\mathsf{noattempts} = \mathsf{odcount}))
\end{array}
$$

Fig. 5. Specification of Account class (part 2)

The n-post-condition withdraw() requires the account not to be frozen at the time of the call to the operation, and the balance in the account at that time to be sufficient. The AccountExc-post-condition on the other hand asserts that this exception will be thrown if the account was already frozen or if the funds in the account are insufficient; this post-condition further tells us that in the former case odcount is not updated when the exception is thrown, but in the latter case it is incremented by 1 (matching our informal understanding of the operation). More interesting is the last line of this post-condition; it tells us that the account attribute of the exception being thrown is (a reference to) the self object, i.e., the account object on which the withdraw() operation was applied; the final clause tells us that the attribute noattempts of the exception is appropriately set. The specification of fwithdraw() is similar but simpler since this operation does not raise an exception even if the balance is insufficient.

We also need to specify the AccountExc class. This is a fairly simple class and its operations just return the corresponding attribute values. We capture this precisely in the OCL specifications for this class in Figure 6.

Ultimately, of course, the purpose of such precise specifications is to help the designer of the client class, Bank in our particular example. Earlier we considered the possibility that this class might be carrying out a series of deposit() and withdraw() transactions on a collection of Accounts. Suppose one of the withdraw() operations applied to one of these accounts raises an exception. The exception

$\underline{\mathsf{AccountExc} :: \mathsf{attempts}() : \mathsf{Integer}}$
$\mathsf{pre} : \mathsf{true}$
$\mathsf{post} : !\{\mathsf{noattempts}, \mathsf{account}\} \wedge (\mathsf{result} = \mathsf{noattempts})$
$\underline{\mathsf{AccountExc} :: \mathsf{acct}() : \mathsf{Account}}$
$\mathsf{pre} : \mathsf{true}$
$\mathsf{post} : !\{\mathsf{noattempts}, \mathsf{account}\} \wedge (\mathsf{result} = \mathsf{account})$

Fig. 6. Specification of AccountExc class

handling-code in the Bank class could be written, in *Java*-like syntax, as follows:

```
catch (AccountExc e) {
        Account a = e.acct();
        if (e.attempts() ≥ 2) { a.freeze(); }
        else ...
    }
```

This catches any exception of type AccountExc; it then checks the number of attempts already made to overdraw funds in this account, and if it is two or more, freezes the account. Note how we are using the information provided by the exception object. According to the specification in Figure 6, the acct() method gives us (a reference to) the account that the exception refers to which, by the specifications Figure 5, is in fact the account that the overdraft was attempted on. The specifications in Figures 5 and 6 also tell us that the attempts() operation will return the number of overdraft attempts made so far on this account. Thus by referring to the information provided in the e-post-conditions of the operations that raise the exceptions, and by referring to the information in the specification of the operations of the exception class, we can, in the client class, make effective use of the information contained in the exceptions.

We conclude this section with a final observation about our example. Consider the 'else' clause in the exception handling code above. What would be an appropriate action to perform in this situation? The natural answer would be to use the fwithdraw() operation to allow the overdraft. We do know the account on which the fwithdraw() operation should be applied; it is the one we got as e.acct(). But what should be the argument to fwithdraw(), in other words, what amount should we force-withdraw? We do not currently have that information. This is in fact, an incompletely designed AccountExc class. A more useful design would have included not just the attributes account, noattempts, but also withdrawAmount, being the amount of money that we were trying to withdraw in this transaction when the exception was thrown. It would be easy to add this attribute, and the corresponding operation wAmount() that returns this value. With this change and the corresponding changes in the specifications if Figures 5 and 6, we will be able to write an appropriate else clause.

5 Discussion

Exceptions are useful not only when implementing OO systems, but also when building business models. The question we have focused on in this paper is, 'What information should we include in our UML models about the exceptional behavior of classes?' Clearly the answer to this question depends on why exceptions occur. A common view of exceptions has been that they represent catastrophic failures such as, for example, power failures, or network crashes, etc. Under this view, the client class, upon receiving the signal that an exception occurred is required to recover as best as it can (or if it cannot, itself fail in turn, and signal its client). A more useful view, both when implementing systems as well as when designing business models, is to think of exceptions as unusual but not disastrous situations. And that exceptions are in fact, full-fledged objects that can be used to convey a range of useful information to the client class. This is the view that languages like *Java* support, and is the view we have tried to model in this paper. The power of this approach is clear even in the relatively simple banking example that we considered. It was only because the AccountExc objects contained information, in particular about the identity of the Account object that was responsible for the exception, that the client class could proceed appropriately once the exception was thrown.

Since exceptions are simply objects, it is easy to represent them in UML class diagrams. We can simply treat them as instances of a stereotype ≪exception≫ with the additional decoration of a caution icon as recommended by [1]. While modeling exceptions in this manner (as in our Figure 3 provides useful information, this is not sufficient. The point is that, by their very nature, exceptions are difficult to understand, and it is important to specify precisely when they might be thrown, and exactly what information will be contained in them when they are thrown. The OCL component of UML is designed to enable precise specification of the behavior of operations of a class. However, standard OCL does not deal with operations that may raise exceptions. In Section 3 of this paper, we proposed a simple extension to OCL to deal with such operations. The main idea underlying our extension was to specify an operation by means of a pre-condition, a normal-post-condition (or n-post-condition), *and* a set of exception-post-conditions (or e-post-conditions), rather than in terms of just a pre-condition and a (normal) post-condition. Each e-post-condition allows us to specify the conditions that will hold if the operation were to terminate by throwing the corresponding exception, just as the normal post-condition allows us to specify the conditions that will hold if the operation were to terminate normally. In Section 4 we saw how this approach could be used to model precisely the behavior of the banking example.

Thus the key contributions of this paper have been to:

- Present a case for why treating exceptions as full-fledged objects rather than just signals is extremely useful not just at the implementation level, but also at the conceptual level.

– Present a simple extension to OCL to allow us to specify precisely the behavior of applications that make use of exceptions in this manner.
– Demonstrate the approach by applying it to a simple illustrative example.

We conclude with a couple of observations. First, UML has other components than just class diagrams and OCL specifications. For example, dynamic views of the system via use-cases and interaction diagrams are extremely important. Suppose an operation is invoked; the invocation is of course represented on the interaction diagram, as is the return. What if the operation were to throw an exception? For a full picture of the system, it is clearly essential to represent this event, as well as the subsequent catching and handling of the exception by the caller on the interaction diagram. It would seem natural to introduce corresponding stereotypical actions to represent these events and use them in the interaction diagram.

Second, we believe that an AccountExc object containing a reference to the Account object that threw that exception is not incidental to this particular example, but rather an instance of a common pattern applicable to exceptions in general. In other words, we believe that it would be useful in general for exception objects to include a reference to the object that was responsible for creating and throwing that exception. Hence it would be useful to include this as part of the definition of the exception-stereotype so this information does not have to be repeatedly included in each class diagram. And in OCL specifications of exception classes, it may be appropriate to use a special notation, such as @source, to refer to the source object. This would eliminate the need for every exception class providing an operation similar to the acct() operation of AccountExc to give us this information.

References

1. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
2. D. D'Souza and A. Willis. *Objects, Components, and Frameworks with UML.* Addison Wesley, 1999.
3. M. Fowler. *UML Distilled.* Addison-Wesley, 1997.
4. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.
5. J. Warmer and A. Kleppe. *The Object Constraint Langauge.* Addison-Wesley, 1999.

This article was processed using the LaTeX macro package with LLNCS style