

# Supporting Evolution of Interface Exceptions

Anna Mikhailova<sup>1</sup> and Alexander Romanovsky<sup>2</sup>

<sup>1</sup> Department of Electronics & Computer Science  
University of Southampton  
Highfield, Southampton, SO17 1BJ, UK

<sup>2</sup> Department of Computing Science  
University of Newcastle upon Tyne  
Newcastle upon Tyne, NE1 7RU, UK

**Abstract.** Interface exceptions (explicitly declared exceptions that a method can propagate outside) are an inherent part of the interface describing the behaviour of a particular class of objects. Evolution of system behaviour is thus necessarily accompanied by and reflected in the evolution of interface exceptions. While evolution of normal system behaviour is adequately supported by various language mechanisms, such as subtyping and inheritance, few contemporary object-oriented programming languages offer support for the evolution of interface exceptions. Some languages permit specialising and deleting interface exceptions while subtyping, but none of them provides an adequate support for adding exceptions. In this paper we propose two complementary solutions for dealing with additional exceptions introduced while system evolution. To solve the problem of non-conforming interfaces resulting from the addition of new exceptions in a development step, the first proposal uses rescue handlers and the second one employs the forwarding technique.

## 1 Introduction

Organising exceptions into hierarchies and specialising exceptions along with the specialisation of classes is in the spirit of the object-oriented paradigm. Few contemporary programming languages support a systematic hierarchical treatment of exceptions in an object-oriented style. We analyse what a more permissive object model supporting evolution of interface exceptions should be like, and propose an improved model, supporting exception addition, that can be incorporated into existing languages.

When specialising a class into a subclass, it is often necessary to

- specialise interface exceptions to subtypes of the exceptions signalled by the superclass
- remove interface exceptions signalled by the superclass
- add new interface exceptions, in addition to those signalled by the superclass

We study in detail these cases, focusing on the semantic implications that they cause in resulting programs. Our analysis of the existing languages supporting an object-oriented style of exception handling, most notably Java [5], Arche

[6], and Modula-3 [2], indicates that, at best, these languages permit specialising and deleting interface exceptions while subtyping, but none of them provides an adequate support for adding exceptions.

We propose two type-safe solutions for the problem of non-conforming interfaces resulting from the addition of new exceptions in a development step. The first proposal is best suited for the top-down system development approach, when we face the need to introduce an interface exception in a development step. The need for introducing a new interface exception may arise, e.g., because a new data structure can deliver new exceptional behaviour. This proposal is based on extending a language with a new construct, a rescue handler, which steps in to rescue the situation when no ordinary handlers are available. Our second proposal is best suited for the bottom-up approach to system development, with which we might want to match an existing class (e.g., from a class library) to an existing interface (e.g., provided by a framework). If the class has extra interface exceptions not signalled by the interface which the class matches otherwise, we propose to employ the forwarding technique, widely used in practical system development to solve the closely related interface mismatch problems.

## 2 Object-Oriented Exception Handling: The Object Model

Exceptions are abnormal events which can happen during the program execution. Most programming languages and systems provide special facilities and language mechanisms for handling exceptions in a disciplined way. More modern object-oriented languages support an object-oriented style of exception handling: they allow arranging exceptions into classes and structuring them into class hierarchies. Apart from delivering better structuring, clarity and conciseness of the resulting code, this approach also promotes genericity and polymorphism, characteristic of the object-oriented style of program development, in the treatment of exceptions.

Recognising the significant advantages of this approach, we also consider the object model where exceptions are class instances, and classes of exceptions are structured into hierarchies. Exceptions can be explicitly created by instantiating the corresponding exception classes and can be initialised using constructors with input parameters. Exceptions can also be created implicitly, when they are raised or signalled. In this case a default (parameterless) constructor is invoked to create an instance of an exception class.

An object, a method, or a block of code can be viewed as an *exception context*, so that developers can declare exceptions and associate handlers with such a context: when an exception is raised in an exception context, the control is transferred to the corresponding handler.

In our view, an important feature of an exception handling mechanism is its ability to differentiate between *internal exceptions* to be handled inside the context and *external exceptions* propagated from the context. These two kinds of exceptions are not clearly separated in many languages, although they obviously

serve different purposes. The separation can be achieved under two conditions: contexts are program units that have interfaces (e.g. classes or methods), and the concept of *exception context nesting* is defined. Most of the existing exception handling mechanisms use *dynamic exception context*, such that the context is the method or the object being currently executed. Some mechanisms use *static exception contexts* based on the corresponding object declaration.

The execution of the context can be completed either successfully or by propagating an (external) *interface exception*. The propagated interface exception is treated as an internal exception raised in the containing context. The simplest example of the dynamic nested context is nested procedure calls. In fact, this is the dominating approach to exception handling which suits well the client/server or remote procedure call paradigms.

In our model, methods are dynamic exception contexts. Each method can be dealing with a set of internal exceptions, each of which must have a corresponding handler associated with the method. Internal exceptions are *raised* in the method code and have to be handled inside the method. Each object type (interface) can have explicitly declared interface exceptions; all interface exceptions that a method can *signal* are to be declared in the method signature using a special signal clause. Interface exceptions are signalled by the method code or by handlers associated with it. Note that interface exceptions of the method called in another method are internal exceptions of the latter and have to be handled at its level. We follow the *termination model* of exception handling [4]. With the termination model, after an internal exception has been handled, the execution of the corresponding method is terminated and the control returns back to the caller.

An example presented in Fig. 1 illustrates the difference between internal exceptions handled by the object's methods and external (interface) exceptions signalled outside the object to be handled by object's clients. The class `Bank` represents banks working with accounts of type `Account`, and its subtypes `CurrentAccount`, and `SavingsAccount`. The method `transfer` of class `Bank` can be used to transfer a certain amount of money from one account to another. If the specified current account `fromAccount` doesn't have enough money, as signalled by its `withdraw` method, an attempt is made to withdraw this amount from a savings account. The method `transfer` has two interface exceptions: `NotEnoughMoneyException` and `SavingsAccountUsedException`. The former is signalled if there is not enough money to be transferred even on the savings account, and the latter is signalled to inform the caller about the fact that the savings account has been used (although the money has been successfully transferred). Internal exceptions `NotEnoughMoneyException` signalled by `withdraw` methods of `fromAccount` and `sAccount` are handled inside the method `transfer`.

Only interface exceptions can be propagated outside the class in our model. All possible violations of this rule must be either detected at compile time or must cause a predefined `Failure` exception to be propagated outside the class. This exception is signalled in some other situations, for example, when it is

---

```

class Bank {
    ...
    public void transfer (CurrentAccount fromAccount, SavingsAccount sAccount,
                        Account toAccount, int amount)
        signals NotEnoughMoneyException, SavingsAccountUsedException
    {
        try {
            fromAccount.withdraw(amount);
        }
        catch (NotEnoughMoneyException neme) {
            try {
                sAccount.withdraw(amount);
                toAccount.deposit(amount);
                signal new SavingsAccountUsedException();
            }
            catch (NotEnoughMoneyException neme) {
                signal new NotEnoughMoneyException();
            }
        }
        toAccount.deposit(amount);
    }
}

```

---

**Fig. 1.** Example of the difference between internal and interface exceptions

impossible to leave the object in a known consistent state corresponding to one of the interface exceptions.

For simplicity, we do not consider multiple inheritance.

### 3 Behaviour Refinement Requires Exception Evolution

#### 3.1 Behaviour Evolution

The evolution of system behaviour is always performed as the evolution of system components. Changes of the component behaviour often cause changes of their interface. Very often the behaviour evolution results in increasing complexity of software, forcing system developers to modify the system structure, to handle this complexity. The most typical way of achieving this is by decomposing some components into several subcomponents. These subcomponents can either be hidden in a higher-level *wrapping* component which conforms to the interface of the original component, or they can themselves replace the initial component and be used by the original component's clients.

There are multiple ways in which the system behaviour can evolve. The most obvious are improving functionality of the components by replacing old fragments of the design, e.g. code, with new better ones (*refinement*), and adding

new functionality (*extension*). Apart from these, there are also other forms of evolution that deserve attention as well: *deleting functionality* and *merging functionality*. These four forms of behaviour evolution cover the main possible directions in which system design can proceed.

Contemporary programming languages provide several language mechanisms supporting behaviour evolution. The principle mechanism supporting behaviour evolution in the context of object-oriented programming is *inheritance*. The classical view is to associate inheritance with *conceptual specialisation* in system modelling and design [8]. This form of inheritance, sometimes referred to as strict inheritance, unifies subclassing (*implementation inheritance*) with subtyping (*interface inheritance*), forcing code reuse and behaviour evolution be necessarily accompanied by conceptual specialisation. Since these two processes are to a certain extent unrelated, this unification appears to be too restrictive for dealing with evolutionary development of complex systems. In particular, the addition of truly new properties requires re-constructing system parts from scratch [8].

To overcome these limitations, the newer object-oriented languages, like Java and Sather, separate interface inheritance responsible for conceptual specialisation, and implementation inheritance dealing with code reuse and behaviour evolution. This results in separate subtyping and subclassing hierarchies. This separation of concepts to a large extent facilitates system design and evolution, because more creative ways of abstraction modification can be explored while subclassing, without the need to maintain behavioural compatibility.

### 3.2 Conceptual Specialisation, Subtyping and Subclassing

Conceptual specialisation, sometime also referred to as *subtyping*, underlies the evolution and behaviour refinement of object-oriented software. Subtyping polymorphism can be used to substitute subtype objects for supertype objects dynamically, at run-time. This permits clients of supertype objects to benefit from conceptual specialisation by using more specialised subtype objects instead of more general supertype objects. For example, method `transfer` of `Bank` can take as argument `toAccount` an object of type `CurrentAccount` or `SavingsAccount`, both of which are subtypes of type `Account` which is the declared type of `toAccount`. Subtyping is usually denoted by `<:`, so that e.g. `CurrentAccount <: Account`, and we will follow this convention here as well.

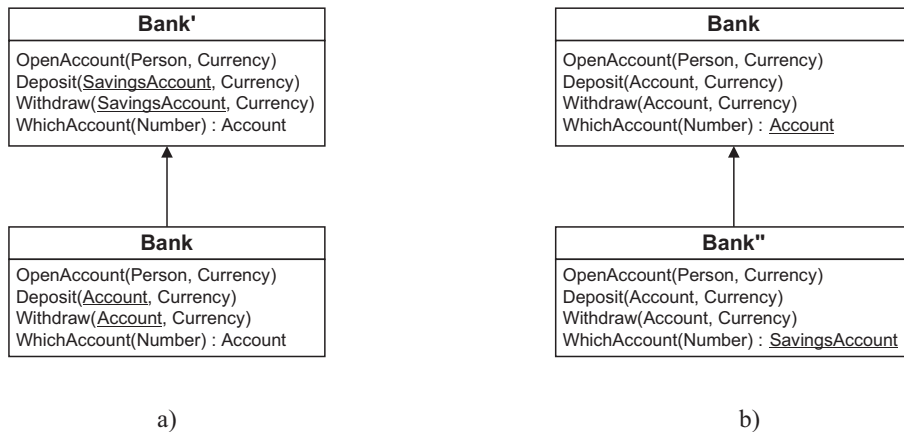
To ensure that all client's requests for method calls on subtype objects can be responded to by supertype objects instead, subtyping requires syntactic conformance of objects' methods. In the simplest case, subtyping is type extension, in the sense that a subtype has all the method signatures of its supertype and possibly also new ones. For example, `SavingsAccount` is a subtype of `Account` if in addition to methods `Owner`, `Balance`, `Deposit` and `Withdraw` of the latter it also has a method `PayInterest` specific to savings accounts.

The subtyping relation, however, does not have to be a simple extension, but can be more permissive in the sense that inherited method signatures can be modified in a subtype so that the types of method input parameters become

*contravariant* and the types of method output parameters become *covariant*. Contravariance means that subtyping on the types of method parameters is in the opposite direction from subtyping on the interfaces having these methods. Respectively, covariance means that subtyping on the types of method parameters is in the same direction as subtyping on the interfaces having these methods. Contravariance in input parameter types and covariance in output parameter types are the basic subtyping properties of function types [1]. As methods are essentially (object state modifying) functions of input parameters returning output parameters, they naturally have these properties as well.

The intuitive meaning of method input parameters is that clients should be able to invoke methods on a subtype object, supplying it with input arguments and obtaining from it results, the same way as they would invoke the corresponding methods on a supertype object. Then input supplied by a client should always be accepted by a subtype method and output produced by the latter should always be acceptable for the client. The contravariance restriction on input parameters and the covariance restriction on output parameters addresses these issues.

Suppose that we have interfaces `Bank` and `Bank'` as shown in Fig. 2(a). `Bank'` has the same method signatures as `Bank`, but with the difference that the methods `OpenAccount`, `ValidAccount`, `Deposit`, and `Withdraw` have the input parameter of type `SavingsAccount` rather than `Account`. As we know, `SavingsAccount` is a subtype of `Account`, and hence a `SavingsAccount` object can be passed where an object of type `Account` is expected. In particular, passing it to the methods of `Bank` expecting an input parameter of type `Account` will be type-correct. According to the typing rules, `Bank` is therefore a subtype of `Bank'`, i.e., `Bank <: Bank'`. Covariance in output parameter types is illustrated in Fig. 2(b). Of the two methods having output parameters, `ValidAccount` has the same output param-



**Fig. 2.** Contravariance in input type parameters a) and covariance in output type parameters b)

eter type in both `Bank` and `Bank''`, whereas `WhichAccount` in `Bank''` has the output parameter type which is a subtype of the corresponding output parameter type in `Bank`. Any output produced by `Bank''` is a valid output of `Bank`, because `SavingsAccount` objects are `Account` objects as well. Therefore, `Bank''` is a subtype of `Bank`, i.e., `Bank'' <: Bank`.

Subclassing or implementation inheritance allows the developer to build new classes from existing ones incrementally, by inheriting some or all of their attributes and methods, overriding some attributes and methods, and adding extra methods.

In most object-oriented languages, such as Simula, Eiffel, and C++, subclassing forms a basis for subtype polymorphism, i.e. signatures of subclass methods automatically conform to those of superclass methods, and, syntactically, subclass instances can be substituted for superclass instances. As the mechanism of polymorphic substitutability is, to a great extent, independent of the mechanism of implementation reuse, languages like Java and Sather separate the subtyping and subclassing hierarchies.

For simplicity, we will consider here subclassing to be the basis for subtyping and will analyse how behaviour refinement of subclasses with respect to their superclasses influences evolution of exceptions. The same principles also apply to systems with separate subclassing and interface inheritance hierarchies, although in these systems subclassing is not necessarily accompanied by behaviour-preserving refinement and can just reflect a behaviour evolution.

### 3.3 Specialising Exceptions

Analysing the nature of interface exceptions, it is easy to see that like method output parameters, they are entities *returned* from a method. As such, like output parameters they are likely to have covariant nature. Indeed, if instead of signalling an exception of type `ArrayException` in a subtype `SortedArray` of `Array`, we will signal an exception `SortedArrayException`, clients using `SortedArray` object and expecting an exception of type `ArrayException` should be able to deal with its special case, `SortedArrayException`. Such covariant exception specialisation ensures that clients using a subtype object instead of a supertype object are never faced with unexpected method results, in this case exception occurrences.

As it is perfectly type-safe to covariantly redefine (specialise) interface exceptions, some languages actually permit this kind of redeclaration. The object-based language Modula-3 was one of the first to introduce some form of interface exception specialisation, although exceptions are not classes here. A procedure declaration includes a list of all exceptions that can be signalled. The language allows procedure redeclaration while exporting interfaces: all exceptions that a redeclared procedure can signal must be declared in the exported procedure declaration.

Method declaration in Java can contain the `throws` clause that has to include all checked exceptions that the method can signal. Java imposes the following rule on the checked exceptions that method `n` overriding method `m` of the superclass can throw: for every exception class listed in the `throws` clause of `n`, either

the exception class or one of its superclasses must be listed in the throws clause of `m`. For example, we can have

```
public interface Buffer {
    void set(char) throws BufferError;
}

public interface InfiniteBuffer extends Buffer {
    void set(char) throws InfiniteBufferError;
}
```

provided that `InfiniteBufferError <: BufferError`.

Naturally, this rule permits specialising one exception class in the throws clause of the parent method to several of its subclasses in the overriding method.

A very similar approach is used for dealing with interface exceptions during subtyping in the programming language Arche.

### 3.4 Removing Exceptions

Apart from specialising interface exceptions while subclassing, some existing programming languages also permit removing them. For example, Java stipulates the “Catch or Specify Requirement” which requires that a method either catches an exception by providing an exception handler for that type of exception, or it specifies that it can throw that exception. What this rule effectively permits is removing in a subclass method an exception signalled by a parent method by handling it internally. An example from [5] illustrates this situation:

```
public interface Buffer {
    char get() throws BufferEmpty, BufferError;
}

public interface InfiniteBuffer extends Buffer {
    char get() throws BufferError;
}
```

It is interesting to note that removing interface exceptions, unlike removing methods, does not restrict the functionality of a subtype. While method removal can by no means be viewed as behaviour-preserving and type-safe, interface exception removing indicates that exceptional or erroneous behaviour is *reduced* in a subtype, and as such can be viewed as behaviour refinement. Clearly, removing interface exceptions in a subtype preserves type safety. Clients using a subtype object instead of a supertype object will never be faced with an exception they are not ready to handle, because fewer exceptions are signalled by the server object. Being prepared to handle the same exceptions as before, the clients will carry out the actual handling less often.

As demonstrated by these examples, the existing languages support covariant redeclaration of interface exceptions and their removal. However, considering general ways in which systems can evolve (Section 3.1), it is clear that these ways of inheriting, redeclaring and removing interface exceptions are too restrictive and should be relaxed to support other forms of behaviour evolution as well.



### 3.5 Exception Inheritance for Exception Evolution

Miller and Tripathi in [7] rightfully point out that the exception handling mechanisms in existing object-oriented languages are oriented towards implementation only and, as such, do not provide an adequate support for system development. We are interested in a mechanism supporting implementation development as well as system evolution. This kind of an exception handling mechanism will help to bridge the gap between different models used at various stages of the software life-cycle and to make the transition between different stages seamless.

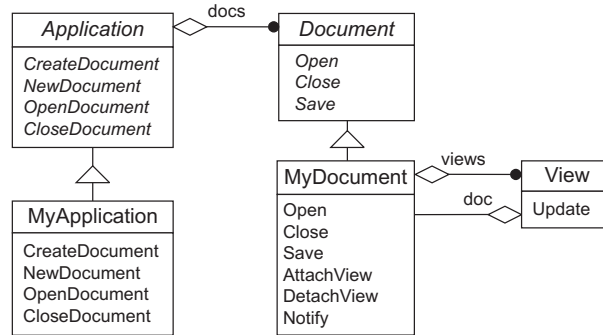
First, we would like to identify the features that an exception handling mechanism supporting various forms of behaviour evolution should possess. For this, let us consider all the possibilities one might potentially like to exercise in re-declaring exceptions when developing a subclass. The existing languages allow specialising exceptions, as discussed above, and removing them. Both forms of exception evolution are useful but insufficient, because they cover only a part of the complete picture. Exception merging is another form of exception evolution. It seems to be possible that at some step of class evolution it will be decided that several independent interface exceptions of a method have to be merged into one exception. This can happen if we find out that they are caused by similar reasons or that we do not want them to be different. For example, heap and stack are usually implemented in the same space but one grows from the bottom and the other one from the top. We may decide to merge the corresponding two exceptions into a single `no_memory` exception if they have to be treated in the same way. Although it may be possible to propose some specialised solutions supporting such functionality, for simplicity we consider that this problem can be solved by deleting exceptions and adding new ones.

### 3.6 New Functionality – New Exceptions

When specialising or extending classes, the existing approaches to dealing with interface exceptions at best permit to specialise and remove superclass interface exceptions in subclasses. However, when developing complex software, developers might be faced with the need to address system evolution requirements for which these interface exception changes are too restrictive.

Consider, for example, the setting illustrated in Fig. 3. Suppose that initially our design consists of classes `Application` and `Document`. An application works with a number of documents and can create new documents, open existing documents and close documents. The correspondingly named methods in class `Application` implement this functionality. A document provides methods that its clients, in particular the application using this document, can invoke to open, save, and close the document. For example, when an application needs to close a specified document, it checks whether the document has been saved since the last modification, saves it if it hasn't and closes the document.

Suppose now that we want one document to be viewed and edited in several windows. To achieve this, we employ the usual Observer Pattern [3], creating new classes `MyDocument` and `View`, such that each `MyDocument` instance can



**Fig. 3.** Example of new functionality requiring new exceptions

be observed by a set of `View` instances. Views can be attached to and detached from a document using the correspondingly named methods of `MyDocument`. Whenever a document is changed in one of the views, it notifies each of its views about the change by broadcasting the method `Update`.

The problem arises when we are trying to implement `MyDocument`'s `Close` method. When an attempt is made to close a document which is simultaneously modified in several windows, we would like to signal an exception `MultipleViewCloseException`. But as method `Close` of `Document` does not signal any exceptions, this redeclaration of its interface in `MyDocument` would be illegal in all the languages supporting only covariant interface exception redeclaration.

As demonstrated by this example, what we would like to have is more flexibility, enabling the kind of interface exception redeclaration when a subtype method can signal *completely new exceptions*. This observation is also made by Miller and Tripathi, who note in [7]: “For exceptions, new functionality may need new exceptions that are not subtypes of exceptions from the parent method”. Further, the authors conclude that “[...] evolutionary program development suggests exception non-conformance”.

Fortunately, this apparently desirable exception non-covariance (or “non-conformance” in terms of [7]) can be successfully dealt with, to circumvent type-theoretic problems. In the following section we present our proposal on how to deal with non-covariant interface exception redeclaration, without sacrificing the type safety provided by the existing exception handling mechanisms. In this manner, a more flexible, yet safe, exception handling mechanism can be built.

## 4 Adding New Interface Exceptions

We envision two closely related ways of dealing with new interface exceptions added in a subclass. The first approach is based on using *rescue handlers* – default handlers attached to the class introducing new exceptions. The second approach employs the forwarding technique.

## 4.1 Using Rescue Handlers

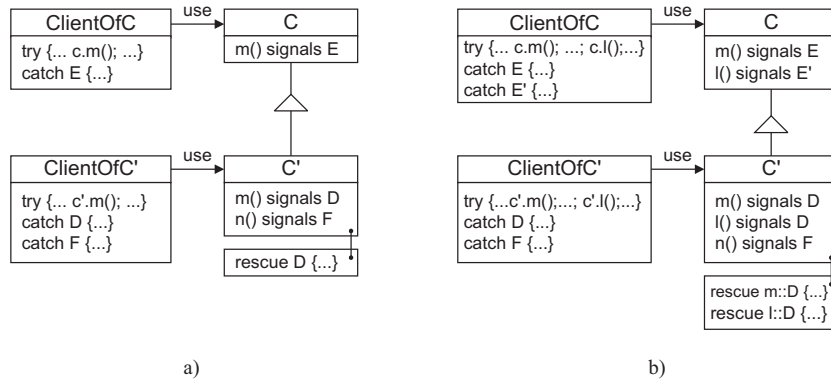
**The General Idea** Consider a class  $C$  and its subclass  $C'$ , which inherits methods of  $C$ , overriding some of them, and adds some new methods. Suppose that a method  $m$  of  $C$  signals an exception  $E$  and its counterpart in  $C'$  signals instead an exception  $D$  which is not a subtype of  $E$ . In addition, suppose that a new method  $n$  of  $C'$  signals an exception  $F$ .

As we know, clients of  $C$  might not be aware of the existence of  $C'$  and the handlers that these clients provide are only prepared to handle the exceptions explicitly declared in the interface of  $C$ . On the other hand, clients of  $C'$  which see the new exception signalled by  $m$  can provide a handler for this exception.

To deal with the new exceptions for which no handlers are available in the client code that invoked the methods signalling these exceptions, we define a default handler – the rescue handler. We chose to call this handler a rescue handler because it is used for the specific situation when clients do not know how to deal with new interface exceptions of their servers, being unaware of their existence, and the rescue handler steps in to rescue the situation. Clearly, this rescue handler should be attached to a server class in which the new exceptions are declared. Of course, it is easy to envision a scenario with which more than one method of a subclass signals the same non-covariant exception; if we have introduced a new data structure or some new functionality in  $C'$  then several of its methods might need to signal the exception  $D$ . In this case, a rescue handler for a new exception signalled by a particular method of a class should be associated with this method. This association of a rescue handler to a particular method rather than to the whole class might be necessary because rescue handling of an exception might require variations depending on the method signalling it. Syntactically, attaching a rescue handler to a particular method will amount to marking the rescued exception with the name of this method. When no ambiguity arises or when one kind of rescue behaviour is satisfactory for all methods signalling this exception, we provide a single rescue clause for each new exception at the class level. We illustrate rescue handlers at both the class level and the method level in Fig. 4. The rest of the discussion applies to both cases.

We view the rescue handler as an auxiliary code executed in the server context when the client does not have the handler for the interface exception signalled by the server. Rescue handlers can manipulate the server state, trying to recover it (possibly with some degradation) or can transfer it to a state corresponding to another server interface exception which will be signalled by the rescue handler.

An important point to note here is that this scenario is type-safe. The client calling a method will never be asked to handle an exception which it does not expect and for which it does not have a handler. The client only gets to handle those exceptions that are declared in the interface of its declared server. The new exceptions signalled by the server's subclass are handled by a rescue handler associated with the server's subclass itself. The task of the compiler is then to check that *every new exception of the subclass has an associated rescue handler attached to the subclass*.



**Fig. 4.** Rescue handlers at class level (a) and at method level (b)

Using this approach, we can now solve the problem in our example of applications and documents. We can allow `MyDocument`'s `Close` method to signal the new `MultipleViewCloseException`, and define a rescue handler for it, attached to the class `MyDocument`. Such a rescue handler can, for example, close all views open on the document and then close the document itself. Then any `Application` instance invoking `MyDocument`'s `Close` method will never be faced with `MultipleViewCloseException` unknown to it: the rescue handler will handle it and return control to `Application`.

Moreover, the clients of `MyDocument`, aware of the fact that the method `Close` of the latter can signal `MultipleViewCloseException`, can handle this exception in a more sensible manner, superseding the rescue handler provided by `MyDocument`. For example, `MyApplication` which works with `MyDocument` directly, rather than via subsumption through `Document`, can define a handler for `MultipleViewCloseException` that will pop-up a dialog inquiring the user whether he really wants to close the document along with all its views, or only wants to close specific views, leaving the document open in the other views.

**Propagating an Exception** Apart from providing some computations attempting to fix the problem, or simply returning the object into a consistent state, the rescue handler can also signal exceptions. Naturally, the exceptions that it can signal must be either subtypes of the exceptions signalled in the corresponding parent method, or they also can be the predefined `Failure` exceptions. More formally, for classes `C` and `C'` such that

```

class C {
    void m() signals E1, ..., En {
        ...
    }
}

class C' extends C {
    void m() signals E'1, ..., E'k, F1, ..., Fm {
        ...
    }
    rescue F1{... signal new H1() ...}
    ...
    rescue Fm{... signal new Hm() ...}
}

```

where  $E'_1, \dots, E'_k$  are subtypes of some exceptions among  $E_1, \dots, E_n$  and  $F_1, \dots, F_m$  are new non-covariant exceptions, every  $H_j$ , for  $1 \leq j \leq m$ , must be a subtype of some  $E_i$ , for  $1 \leq i \leq n$ , or Failure:

$$(\forall j | 1 \leq j \leq m \cdot (\exists i | 1 \leq i \leq n \cdot H_j <: E_i) \vee H_j = \text{Failure})$$

This rule extends to hierarchies of larger depth in an obvious manner, recursively: if a method  $m$  defined in a subclass  $C''$  of  $C'$  signals an exception  $G$  which is non-covariant to either of  $E'_1, \dots, E'_k, F_1, \dots, F_m$  then a rescue handler for this exception defined in  $C''$  can only throw exceptions that are either subtypes of  $E'_1, \dots, E'_k, F_1, \dots, F_m$  or Failure.

**Implementation Details** Let us consider now how our proposal can be implemented in practice; in particular, how the control is passed at runtime between client objects and supplier objects signalling new exceptions. Two general scenarios are of interest here:

1. The client is not aware of the new exceptions and the rescue handler is to be invoked
2. The client is aware of the new exceptions and its own handler is to be invoked, superseding the rescue handler.

Suppose that we have a certain class `NewSupplier` extending some parent class `Supplier` and overriding a method  $m$  of the latter so that it signals a new (non-covariant) exception  $E$ .

```

class NewSupplier extends Supplier {
    void m() signals E {
        try {
            S1;
            signal new E();
            S2;
        }
        catch (-internal exceptions-) {-handle internal exceptions-}
    }
    ...
    rescue E {RE}
}

```

Suppose also that we have two clients for `NewSupplier`, the one using it through subsumption and unaware of the new exception `E` (we will call it `Client`), and `NewClient` which knows that it uses `NewSupplier` and is prepared to deal with its new exception.

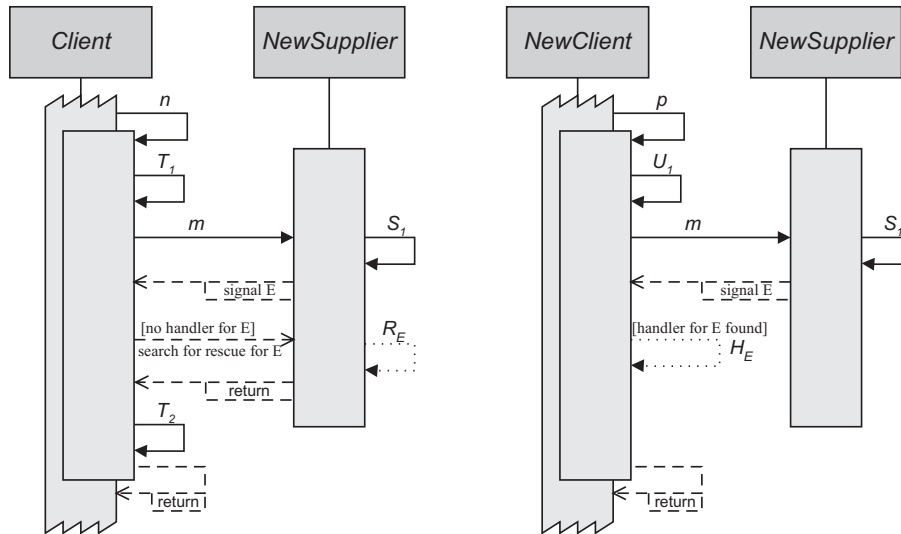
```

class Client {
  void n() {
    try {
      T1;
      s.m();
      T2;
    }
    catch B {HB}
    catch C {HC}
    ...
  }
  ...
}

class NewClient {
  void p() {
    try {
      U1;
      s.m();
      U2;
    }
    catch D {HD}
    catch E {HE}
    ...
  }
  ...
}

```

We illustrate the control flow for both scenarios in Fig. 5, using sequence diagrams. As usual, the vertical dimension represents time and the horizontal dimension represents the actors involved in a collaboration; time proceeds down the page. Solid arrows denote method invocations and ordinary actions, like assignments and iterative statements; dashed arrows denote control passing between the actors involved; finely dashed arrows denote exception handler invocations.



**Fig. 5.** Control flow for clients invoking a method signalling a new exception

As shown in this diagram, when the method `n` is invoked on `Client`, the first action to be performed is  $T_1$ . For simplicity, we have shown this action as the one performed on `Client` itself, but in reality it can be something more involved, like a sequence of method invocations. The invocation of `m` on `NewSupplier` results in transferring control to the latter, which executes  $S_1$  and then can signal the exception `E`, which is new and unknown to `Client`. The control is passed to `Client`, which searches for a handler for `E` and, having not found one, returns the control back to `NewSupplier`. The latter searches for a rescue handler for `E`, and having found  $R_E$  executes it. Provided that  $R_E$  successfully fixes the problem, the control is returned back to `Client` which executes  $T_2$  and returns control to the client which invoked method `n`. In case the rescue handler  $R_E$  itself signalled an exception, this exception is propagated to `Client` which reacts to this exception in the usual way, handling it or propagating it further. Recall that all exceptions signalled by  $R_E$  are required to be either covariant to one of the interface exceptions of `Supplier`'s method `m`, or the predefined `Failure` exception.

Consider now the collaboration between `NewClient` and `NewSupplier`. `NewClient` is aware of the possibility that method `m` of `NewSupplier` signals `E` and is prepared to handle it. When `E` is indeed signalled, `NewClient` catches it and invokes the handler  $H_E$ . This handler supersedes the rescue handler provided by `NewSupplier`. It is interesting to note that, conceptually, the “ordinary” handler defined in the client overrides the rescue handler in the server, although they are located in different classes.

Strictly speaking, with (successful) rescue handling we deviate from the termination model of exception handling employed elsewhere in our model and use the resumption model instead. The reason for this is that exception handling takes place in the server context rather than the client context. If the server itself has managed to correct the problem in the associated rescue handler, it terminates normally and returns control to the client. There is no need to terminate the client which can be left unaware of the exceptional situation that has been successfully resolved and just proceed normally.

As we already mentioned above, our solution to the problem of new exception introduction is type-safe. The type safety is imposed through requiring that a compiler verifies that every new exception of a subclass has an associated rescue handler attached to the subclass. To enforce this safety rule, we can always provide a default rescue handler signalling `Failure`.

**Inheriting Rescue Handlers** When subclassing a class providing rescue handlers for new exceptions, the rescue handlers are inherited and can be overridden. When no new rescue clause is provided in a subclass, the one from the parent method is inherited. To override a rescue clause for a particular exception, the subclass should simply provide a new rescue clause for this exception. There is no need to delete rescue clauses in a subclass, because even if we drop the interface exceptions for which rescue handlers were defined in a superclass, no harm is done if these handlers are inherited.

An important special case of this rule is applied when a new exception is covariantly redefined in a subclass of a subclass signalling and rescuing the new exception. The designer of such a subclass may choose to either use the existing rescue handler for the corresponding superclass exception or to develop a new rescue handler. For example, if a subclass  $C'$  of  $C$  signals a new exception  $E$  and provides a rescue handler  $R_E$  for it, then a subclass  $C''$  of  $C'$  signalling instead a subtype  $E'$  of  $E$  can either inherit  $R_E$  or redefine it with  $R'_E$  that can be better suited to rescue  $E'$ . By default, when no new rescue clause is provided, the one from the parent method is inherited.

## 4.2 Forwarding to the Rescue

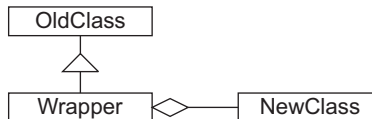
Using rescue handling to solve the problem of new interface exceptions is perfectly suitable for the top-down system development approach, when we face the need to introduce an interface exception in a development step. As discussed above, the need for introducing a new non-covariant interface exception may arise because a new data structure can deliver new exceptional behaviour.

However, rescue handling is of little help if we are to use a bottom-up approach to system development. With this approach, we might want to match an existing class (e.g., from a class library) to an existing interface (e.g., provided by a framework). It is quite likely to happen that the class has extra interface exceptions not signalled by the interface which the class matches otherwise.

To reiterate our example of applications and documents, suppose that the class `MyDocument`, described above, is supplied by a certain class library. Suppose also that we have an object-oriented framework containing an interface `Document` with methods `Open`, `Close` and `Save`. The class `MyDocument` almost exactly matches the interface `Document`, except for the `MultipleViewCloseException` signalled by its method `Close`.

Fortunately, architectural solutions that have proven their usefulness in solving closely related interface mismatch problems, literally speaking, come to the rescue in this situation as well. In particular, *forwarding* or the Wrapper Pattern [3], is an architectural solution that allows clients using instances of `NewClass`, which is an improved, more specialised version of some `OldClass`, but with a slightly mismatching interface, instead of instances of `OldClass`.

The idea behind forwarding is to introduce a subclass of `OldClass`, `Wrapper`, which aggregates an instance of `NewClass` and forwards `OldClass` method calls to `NewClass` through this instance. We illustrate this forwarding scheme in Fig. 6.



**Fig. 6.** Forwarding `OldClass` method calls to `NewClass` via `Wrapper`



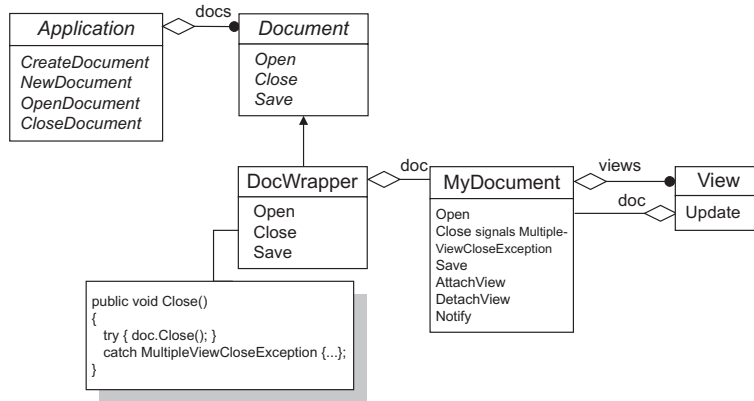


Fig. 7. Forwarding Document method calls to MyDocument via DocWrapper

We can apply the same approach to solving the problem of mismatching interface exceptions, if we turn the new interface exceptions of `NewClass` into internal exceptions of `Wrapper`. The latter, having the same (or conforming) interface as `OldClass`, simply forwards all method calls to the corresponding methods of `NewClass`, catching and handling all `NewClass`'s interface exceptions that cause the interface mismatch with `OldClass`. With this approach, clients of `OldClass` can effectively use `NewClass`, without being concerned that the latter signals an exception of which they are unaware.

In our example of applications and documents, we can solve the problem caused by mismatching interface exceptions in the class `MyDocument` as illustrated in Fig. 7. The class `DocWrapper` implements the interface `Document` by aggregating an instance of class `MyDocument` and forwarding all method calls to the corresponding methods of `MyDocument`. The method `Close` of `DocWrapper` is defined to forward the method call to `MyDocument` and catch the `MultipleViewCloseException` that the latter can signal.

As `Wrapper` classes are just ordinary classes, they can be extended and reused in the usual way.

The two approaches to handling new interface exceptions, the one employing rescue handlers and the one using the forwarding technique, nicely coexist, complementing each other. If a class provides rescue handlers for some of the interface exceptions signalled by its methods, and in addition the application using this class provides a wrapper class catching and handling these exceptions, then the wrapper's handler supersedes the rescue handler provided by the class.

## 5 Conclusions and Future Work

There is a significant gap between methods used for system modelling and design at the earlier phases of the system development life cycle and the methods and

mechanisms supporting the implementation development. One of reasons is a different view these methods and languages have on the way interface exceptions can evolve. In particular, none of the existing programming languages allows adding interface exceptions, which is vital for adding new functionality during system evolution. In this paper we have proposed two type-safe approaches which can be introduced into object-oriented languages to make it possible to add interface exceptions during subclassing. Our future research will focus on further development of these ideas. The intention is to apply these features in design and implementation of several case studies, to analyse possible implementations of these language mechanisms and their overheads, and to propose a formalism for reasoning about systems containing subclasses which have new exceptions and which employ our approaches for dealing with them.

## Acknowledgements

We would like to thank Christophe Dony, Ricardo Jiménez-Peris, Jørgen Lindskov Knudsen, and Anand Tripathi for valuable comments and constructive criticism. Anna Mikhailova's research is supported by the European IST project MATISSE and Alexander Romanovsky's research by the European IST DSoS project.

## References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. Technical Report 52, Digital Equipment Corporation, Systems Research Center, 1989.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
4. J. B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, Dec. 1975.
5. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Sun Microsystems, Mountain View, 1996.
6. V. Issarny. An exception handling mechanism for parallel object-oriented programming: towards the design of reusable, and robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–40, 1993.
7. R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, New York, NY, June 1997.
8. A. Taivalsaari. On the notion of inheritance. *Comp. Surveys*, 28(3):438–479, September 1996.