CLASS AND OBJECT ATTACHED EXCEPTION HANDLERS

A. Iliasov
Faculty of Computer Science, Kyrgyz-Russian Slavic University
Kyrgyz Republic, Bishkek
E-mail: alex@iliasov.org

This paper describes an approach to extending the exception handling mechanisms of object-oriented languages by attaching exception handlers to classes and objects. Object and class exception handlers bring advantages of the object-oriented programming into error recovery actions and give opportunity to consider exception handling policy at the early stages of the abstract class design. Special emphasize is given to the realistic and efficient implementation suitable for statically-typed compiled languages. Discussion is based on Lemick language. The described extension is orthogonal to the existing exception handling mechanism and does not require any changes to language syntax or grammar as it is fully expressed in terms of object-oriented programming already present in the language.

1. Introduction

An exception is the indication of an abnormal situation that may occur during program execution. For languages without exception handling mechanism handling an exception means placing additional code for the detection of an erroneous condition and passing information about the exception via return value. Global variables were often used to save additional information about the exceptional situation. To make exception handling more efficient, languages like C used dangerous constructions like setjmp/longjmp and even pieces of assembly code. This made code hard to read and maintain and thus was an additional source of problems. Exception handling is a language extension specifically aimed on providing high-quality readable code with structured error recovery actions. An exception in object-oriented programming is an object describing an exceptional situation occurred somewhere in program. Such exceptions can be organized into inheritance hierarchies and extended to bring additional information. Exception handlers may also follow exceptions hierarchy and provide common actions for families of exceptions.

One major advantage of exception handling is that it provides a clear separation between the normal program code and code used for error recovery by aggregating all the recovery activities into exception handlers. These handlers are associated with lexical blocks or language structuring units, like methods, classes, modules and objects. Most languages with exception handling capabilities provide lexical block handlers as the only way of attaching recovery code (C++, Java). However such handler binding does not fit well to structuring used in object-oriented programming. The aim of this work is to outline an approach to associating exception handlers with classes and objects and to discuss possible implications for the language, the compiler and the run-time system.

The further discussion is based on Lemick programming language project [1]. It is a statically-typed language compiled into platform-independent virtual machine assembler that is later transformed into a platform-dependent representation by just-in-time compiler. Lemick has a support of concurrent and distributed programming (multi-threading and distributed multi-threading). Rendezvous are used for message passing; replicas and ultra-weak consistency model implement distributed shared memory simulation. Extension of exception handling mechanism, including distributed version is being developed now.

Lemick currently implements a standard approach to exception handling – handlers are attached to lexical blocks and exceptions are objects. In Lemick any object can be raised or signaled as an exception. Signaling means routine abort without trying to handle the situation within the current methods. Raised exceptions may be handled within the method or signaled outside if there is no appropriate handler. Methods and classes may bring signature of the signaled exceptions. Lemick associate exception handlers with blocks of code using the *try-catch* construct, only the termination model is supported [2]. Syntax for *try-catch* is the given on fig. 1.The *try-catch* blocks can be nested and the *catch* blocks must be arranged so that
`ExceptionClass1 <= … <= ExceptionClassN`
Where `<=` holds if a class on the left side is either a subclass or not relative of a class on the right side. All classes are subclasses of `Object` class, so catch-all rule may be the following:

```
        catch e as Object.
```

```
try
    … ‘ protected region
catch a as ExceptionClass1
    … ‘ handler for ExceptionClass1
catch a as ExceptionClass2
    … ‘ handler for ExceptionClass2
...
catch a as ExceptionClassN
    … ‘ handler for ExceptionClassN
finally
    … ‘ clean-up action
end try
```
fig. 1. try-catch-finally syntax

Two built-in classes describing exceptional objects are provided for convenience: `Exception` and `InternalError`

Though they contain the same members and methods, formally there is no relationship between them (in terms of inheritance). `InternalError` and all its subclasses are specific in that they are ignored during compile time reliability checks and they can be omitted in methods signature even if explicitly signaled. Run-time environment knows about these classes and is able to create instances of them. Applications are expected to extend the `Exception` class when they use exception handling for error recovery since for an unhandled exception, run-time environment may choose some default action. However it is not required to raise or signal solely instances or derivates of the `Exception` and `InternalError` classes, any class instance can be raised or signaled. This helps the developers to apply exception handling for the purposes not directly related to error recovery.

2 Class handlers

Class handlers are class-attached exception handlers that can handle one or more exceptions for all the instances of the class. As a class handler we may require an implementation of some fixed interface containing a single public method with the following definition:

```
final interface StaticClassHandler
    declare public sub ClassHandlerStatic(e as Object)
end interface
```

This approach allows definition of only one exception handler that is basically a method in the given class. The only argument of this method is the raised object. The method can handle the situation and exit normally or do abnormal return using **`signal`** statement. Though such handler is not typed, unlike **`catch`** clause, implementation of **`catch`**-like behavior is rather straightforward:

Implementation of the class handler

```
public sub ClassHandlerStatic(e as Object)
signals[1]
        select case e
                case is is[2] MyException1
                        ...   ‘ do something
                case is is MyException2
                        ...   ‘ do something
                case else
                        signal e
        end select
end sub
```

Corresponding **try-catch** syntax

```
try
        ...   ‘ do something
catch e as MyException1
        ...   ‘ do something
catch e as MyException2
        ...   ‘ do something
catch e as Object
        raise[3]
end try
```

For complex classes, which derive part of their functionality from parents, it is possible to use the parent’s handlers[4]. For example, the catch-all action in the code above may be substituted with a call to the parent’s handler, which was overrided[5] by the current handler:

```
catch e as Object
        SomeParentClass.ClassHandlerStatic[6] this, e
end try
```

3 Object handlers

Attaching exception handler to an object means changing object properties or state during execution, since an object handler is specific for the every object instance. This is done by defining some (public) class property that describes the attached handler. This description could be an instance of the following abstract class:

```
class ExceptionHandler
    declare public sub Handler(e as Object)
end class
```

Classes, for which instances we would like to have object-attached handlers, could be derived from the following class:

```
class DynObjectHandler
    public ObjectHandler as ExceptionHandler
end class
```

Using this class is quite easy - we create an instance of a class derived from `DynObjectHandler` and then assign the desired object handler. The definition above allows re-attachment of handlers in any moment at any place, since `ObjectHandler` property is declared public. To restrict access to this property it may be declared private or protected. For the case with private access modifier, handler must be attached inside of a class constructor. Alternatively object-attached handler support can be provided by implementation of interface with getters equivalent to the property declaration:

```
final interface DynamicObjectHandler
    declare public property get Handler as ExceptionHandler
```

[1] it is the same as **signals Object**. It means that routine may signal anything.

[2] **case is** construct expects binary operator name after it. operator **is** tests whether given object is an instance of the given type.

[3] **raise** with an argument is not permitted inside catch blocks, as well as any form of **signal**. **raise** without arguments means re-raising in an outer scope.

[4] applicable only to languages with single inheritance, such as Lemick in our case

[5] note that `ClassHandlerStatic` was not declared as static (and it is also not final). Logically handler is the same for all class instances and therefore it could be static. Unfortunately static methods can be overrided.

[6] though it may look like a static method call, it is actually a special syntax used to call non-static overrided methods

```
    declare public property get Handler(e as Object) as ExceptionHandler
end class
```

In this case we explicitly request an object to give run-time system a description of an exception handler; the object can examine its current state and even the raised exception (for the second method prototype) and choose the best handler from the internal collection of handlers. Class constructors or some other methods should configure initial internal state of the newly created object.

4. Adding support for class and object attached handlers

The described handler attachment scheme is useful only if exception binding logic knows about it and is able to get use of it. Implementation of exception handling mechanism is rarely discussed as it often uses low-level details and system-specific features. That is also true for Lemick handlers binding algorithm:

```
               a method source                               VM assembler
------------------------------------------------------------------------------------
                                                               enter      #96, 0
                                                             ehtable      L#97
                                                   l99:
 sub foo                                                        objvi      #24, 8, r0p
        try                                                      push      r0p
                raise new Exception                              epop
        catch e as Exception                                    raise
        end try                                    l100:
 end sub                                                         jump      L101
                                                   l98: l101:
                                                              signal
                                                   l96:
                                                               leave
                                                   l97: 1  4  99 100 101 1 5 98  ; EH table
```

The exception handler search procedure:

1. Look for the first try block containing the instruction which has raised an exception, compiler must place try-catch blocks so that the enclosing try-catch will always follow the enclosed one.
2. For every catch clause in the selected try-catch block check if the raised exception is an instance of the class associated with the catch clause.
3. If matching catch found then it is invoked, at its end catch clause may either leave the whole try-catch block or propagate an exception in an outer scope.
4. If no matching catch found in the current block then continue search in the enclosing try-catch block
5. If an exception is not handled within the current method, the method is aborted and the exception is propagated to the caller's context.

Now let us look what additions are required to support class and object handlers.

```
 class source with class and object attached handlers          VM assembly for method foo()
-------------------------------------------------------------------------------------------
 class ProtectedClass
 extends DynObjectHandler
 implements StaticClassHandler                                 enter   #120, 0
        ' assign object handler in the constructor             loadc   sp[-4]
        public sub New(OH as ExceptionHandler)                 objvi   #24, 8, r0p
                ObjectHandler = OH                             push    r0p
        end sub                                                epop
        ' this is a class-attached handler (empty for brevity) signal
        public sub ClassHandlerStatic(e as Object)             l120:
 signals:end sub                                               leave
        public sub foo signals Exception                       l97: … ; EH table
                signal new Exception
        end sub
 end class
```

`loadc` is a new instruction that loads **`this`** value (pointer to the current object instance in a non-static methods) into internal hidden field. Now we have enough information

to include search for class and object handlers into handlers binding logic. Additional handler binding steps are the following:
Step 0 is added before step 1 to look for an object-attached handler:
0a. If **`this`** *values is not loaded then proceed with step 1*
0b. if **`this`** *is instance of* `DynObjectHandler` *then cast* **`this`** *to type* `DynObjectHandler`*, check if* **`this.`**`ObjectHandler` *is not null and invoke* **`this.`**`ObjectHandler.Handler(e)` *where* `e` *is the raised exception.*
0c. Examine handler's return context, if the exception is handled then exit binding procedure and continue execution after the point where the exception was raised, otherwise proceed with step 1

Step 5 is extended with lookup for a class handler before aborting the method:
5a. If **`this`** *values is not loaded then proceed with step 5*
5b. If **`this`** *is instance of* `StaticClassHandler` *then cast* **`this`** *to* `StaticClassHandler` *and invoke* **`this.`**`ClassHandlerStatic(e)` *where* `e` *is the raised exception*
5c. Examine handler's return context, if the exception is not handled then propagate it to the caller's context, otherwise abort the method with predefined `ClassHandledFailure` *exception.*

Note that class handler does not do complete error recovery since method is aborted without returning proper result. However successful execution of the class handler guarantees that the object is in valid state and it is safe to operate on it.

5. Discussion

This work follows the general ideas of introducing class handlers presented in [3] and [4]. Work [3] describes object-oriented exception handling build on the top of an existing dynamically-typed language using its reach reflection features. Unfortunately the proposed ideas can not be directly applied to statically-typed languages such as C++ or Java. The same holds for Beta language [6] which employs a class attached and a number of other static handlers. Paper [4] suggests organizing objects into hierarchies to facilitate exception handling activities and discusses an extension of C++ for these purposes. Class level handlers are described with a binding procedure similar to those given in our work; however the proposed exceptions representation and proposed syntax of extensions seem to contradict the existing C++ design. Also it is not clear how the proposed model relates to the existing C++ exception handling mechanism.

Acknowledgments

I would like to thank Alexander Romanovsky for his useful comments and suggestions.

References

1. Lemick project website: http://lb.to.kg/
2. Goodenough, J.B.: Exception Handling: Issues and a Proposed Notion, *Comm. ACM*, 18(12), 1975.
3. C. Dony. A fully object-oriented exception handling system: rationale and Smalltalk implementation. Springer Verlag , 2001, pp. 18-38 In Advances in Exception Handling Techniques (LNCS-2022), Eds. A. Romanovsky, C. Dony, J.L. Knudsen, A. Tripathi
4. A.B. Romanovsky, I.V. Shturtz, V.R. Vassilyev. Designing fault-tolerant objects in object-oriented programming. In Technology of Object-Oriented languages and Systems - TOOLS 7. Ed G. Heeg, B. Magnusson, B. Meyer. 7th International Conf. on Technology of Object-Oriented Languages and Systems (TOOLS EUROPE'92), Dortmuth Germany - TOOLS V. 7. Prentice-Hall. 1992. pp.199-205.
5. A.F. Garcia, C. M. F. Rubira, A. Romanovsky, J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Journal of Systems and Software. 59, pp. 197-222, 2001.
6. J.L.Knudsen: Exception Handling and Fault Tolerance in Beta. Advances in Exception Handling TechniquesSpringer-Verlag, 2001.