Towards an exception handling mechanism in object oriented design

C. Bamford & M. Ramachandran

School of Computing and Mathematical Sciences, Liverpool John Moores University, Liverpool, L3 3AF, UK

Abstract

The detection and handling of exceptions is essential to the design and implementation of reliable and reusable systems, but existing OOD methods do not provide explicit support for them at the design level. We propose an OOD model based on design for reuse with exceptions. Our approach is to categorise the possible exceptions associated with the methods of a class, with the possibility of relating then directly to formally specified pre- and post-conditions. In our work, an exception is treated as an object of a general exception class which acts as a superclass for all specific exception classes. A prototype model of a tool support is proposed, which can provide advice and analysis on the application domain knowledge in designing reusable classes with exceptions.

Introduction

OOD has emerged to support software reuse, software quality and maintaining large software systems. To achieve quality in a design we need to consider a number of design criteria as early as possible in the design process, namely design for reuse, exceptions, and exception handling mechanisms. Existing OOD approaches do not consider these issues early in the design process [1-5]. It is often assumed that the developed OO systems are reliable, reusable and adaptable.

We argue that it is essential to consider these issues and design for them. Our previous works have concentrated on designing OO components for reuse [6-8]. In this article we concentrate on how to identify and handle exceptions in OOD. There have been several interesting works on exception handling at the programming level [9-17], but none of them have considered these as a design issue. We propose an object-oriented exception handling mechanism, known as *OOD for reuse with exceptions*, which is a process based on a number of design stages.

The concept of an exception was introduced in the study of fault-tolerant systems, Goodenough [11], where it is defined to be an unusual situation that needs to be handled differently from normal cases. As such, exceptions cover error cases and certain other cases which are not errors but need separate treatment.

The need to provide special treatment for exceptions is evident, but traditional design and implementation has not separated the treatment of normal cases and exceptions. The separation can be achieved by providing exception-handlers for the exceptions and having the normal processing invoke them when it detects that an exception has occurred. The invocation is called "raising" or "signalling" the exception. The exception-handler takes the appropriate action for that case, such as reporting the condition or executing recovery steps, and either returns to the invoker or terminates.

Research on exceptions at the programming level has discussed the mechanisms needed at run-time. Both the continuation and termination models have been examined. In object-oriented programming, where objects may be created by the normal processing before an exception is identified, it is important to be able to release the resources they use even when the normal processing cannot be continued.

Support for exceptions in programming languages is patchy. Many languages (e.g. C, C++, Pascal) provide no support, but others (e.g. Ada, PLI, CLU) do. In Ada, exception names are declared using the built-in type *exception*. A program unit which detects an exception situation may raise the exception. This transfers control to an exception-handler which may be defined at the end of the program unit in an exception section, or perhaps in a higher-level program unit. The exception is automatically propagated to the higher-level unit if no handler is found for it at the lowest level.

Proposals to extend C/C++ to incorporate exception-handling have been reported, Gehani[16], Stevens[13]. The latter introduces a function called throw to raise an exception and pass a parameter. It is matched with a function called catch defined with the same type of parameter, which activates the *exception-handler*. Dony [9] has described an exception-handling system for an OO language Lore, designed to implement fault-tolerant systems.

The research has concentrated on the programming level and not considered the problems at the design level. For effective use to be made of the separation of exceptions from their handlers, they must be considered early in the development, at the design stage.

The novelty of our approach to exceptions are,

- Identifying and analysing exceptions as a part of the design issue (during OOD),
- Decomposing into a number of domain-specific reusable classes on exceptions and handlers,
- Exceptions based on pre- & post-conditions of each object operation,
- Using the application domain knowledge for the identification of exceptions,
- Tool support which can provide advice and analysis on exception mechanism.

Exception handling in object-oriented design

In the OO context, an exception occurs when an unusual situation arises during the execution of some method of a class. The situations which give rise to exceptions can be broadly categorised as follows:

- run-time support exception, such as hardware or system software failure;
- exceeding limitations of implementation, such as overflow;
- violating the method's pre-condition;

All software relies at run-time on the support of the system software and hardware. If an unusual situation arises in this environment, normal processing may not be able to continue immediately and the exception will need to be handled separately. Certain exceptions can be detected by the software, such as file access failure, and action can be taken by the application. Others, such as a memory failure, might be better left to a global handler to report.

Limitations in the size of data items, such as single-word integers, can cause exceptions when the actual data is larger than the implementation allows. These are hard for the software to detect, and best left to a global handler to report. Exceeding constraint bounds on the allocated size of arrays is easier to detect, and it may be desired to have the application handle the exceptions.

The pre-condition of a method is the condition that is required to be true of the current values used by the method when it begins. For example, the method to Pop an item from a stack object needs the current value of that stack to be non-empty. The corresponding post-condition is the condition on the values which will be true when the method terminates. When formal methods are employed, the techniques of formal verification may be used to show that a method will terminate with a state satisfying the post-condition whenever its pre-condition is true at the start. Even when no formal techniques are applied, the method is designed to work for all initial states satisfying the pre-condition.

If the pre-condition of the method is violated, the method is being used in a situation it is not designed to handle. The normal processing will not be appropriate and special processing will be needed. The exception mechanism can be employed to raise the exception as soon as it is known that the pre-conditions have been violated. Note that it may not be clear at the start of the method that this is the case. Checking a complex condition may be as difficult as carrying out the normal processing, so it may be better to start the normal processing and detect the exception when it becomes clear.

The separation of normal cases and exception cases at the specification level is a well-liked feature of the formal specification language Z. It uses schemas to define operations, with the initial and final values of the state clearly distinguished. Conditions are used to state the requirements of the operation and the pre-condition can be derived from them. A robust operation is often defined as a combination of two or more operations, one defining the normal cases, and others catering for exceptional cases which do not satisfy the pre-condition for the normal cases.

Designing a class for reuse in a particular domain has additional problems as far as exception-handling is concerned. Although the detection of exceptions is associated with the class, the ways in which those exceptions will be handled may depend on the particular application. In general, designing for reuse implies keeping the design as flexible as possible, which in this case means finding a way to be flexible about the provision of the handlers.

The method we propose is to treat an exception as an object. When an exception situation is detected, the exception is raised, which creates the object for that exception and acts as the message to invoke an exception-handler. As far as the class is concerned, it is dealing with exception objects which belong to a superclass for all exceptions. This superclass is similar to the notion of a generic parameter.

When the class is to be used in an application, a specific exception class may be defined for the exceptions. It is here that the exception-handlers are defined. The instantiations of the reusable class are defined by associating the specific exception classes with the exceptions.

Suppose, for example, that a reusable class X is being defined with two public operations, Xput and Xget. If there are two possible exceptions, E1 and E2, which can occur when Xput is executed, and one, E3, which can occur when Xget is executed, the definition of the class will declare three exceptions:

E1, E2, E3 : **exceptions**;

The implementations of the methods for Xput and Xget, will have steps to raise the exceptions when they are detected:

```
Xput:            Xget:
 ...         ...
 raise(E1)        raise(E3)
 ...         ...

 raise(E2)
 ...
```

These are designed and implemented without knowledge of the precise details of the exception-handlers.

When it comes to reusing the class X for an application, a specific exception class containing the handlers for the three exceptions E1, E2, E3 can be defined. If this is called SP, the instantiation of the class X can be referred to as X(SP). With the three exceptions of X having handlers defined in SP, the class X(SP) has handled them all locally, so they are not exceptions for the instantiated class.

If an exception in the reusable class has no handler defined for it locally, it remains an exception for the instantiated class and can be propagated up to the invoking method which may be of a different class. An exception-handler may be defined for it there, or at a higher level. If no handler is defined for it anywhere, a default global handler will be applied.

At design time, it is necessary to identify the possible exceptions from lower-level methods, to determine which should be handled locally, and which passed up to higher-level operations.

A handler is defined as part of the raise method in a specific exception class. However, it may need to have access to details about the object whose method raised the exception. At first it seems that a simple solution would be to pass the details as additional parameters in the raise, but that assumes the invoking method knows what to send. Instead, we propose that the exception class is treated specially, and the exception handler is automatically sent a reference to the invoking object. The handler can then send messages to the object to retrieve the details it needs.

The public interface of the class will contain all the operations that normal access to the class requires. The exception-handlers are unusual and may want further operations. Two possibilities suggest themselves: extend the public interface to let the exception-handlers have access to what they need, or keep the public interface the same but give exception-handlers access to all private details. Neither is attractive. The first may give undue public access, losing the information hiding principle, while the second may give exception-handlers too much access. As a third option, we propose that in addition to the public and private sections in a class definition, there is a section for exception-handlers, with access operations defined for them, but not generally available to the public section.

A case study

The front-end and back-end concept of a compiler design is an illustration of design for reuse. This concept allows you to generate code for different languages. Consider an object-oriented view of a compiler as shown in Figure 1, which consists of high level decomposition of subsystems namely lexical analyser, syntax/semantic analyser, common exception class, and domain-specific exceptions.

The common exception class encapsulates exceptions relevant to hardware, operating systems, user interface, program error, input and output error, process error (in the case of concurrent and distributed applications), object control error, and other environments. The domain exception class encapsulate exceptions relevant to the compiler components such as lexical analyser, grammar specification, syntax/semantic analysis, parsing, and so on.

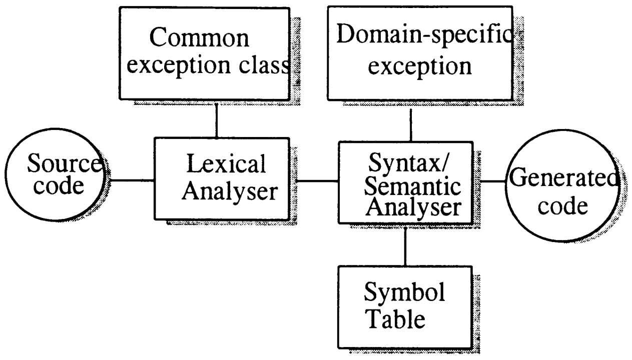

Figure 1 An object-oriented view of a compiler

As a small case study, we shall examine, in outline, the design of a reusable class for the lexical analysis phase of compilers/interpreters for a family of languages. The problem arose as part of another research project we are undertaking into formal languages.

The compilers/interpreters for the languages all share the task of recognising lexical tokens from an input stream. The lexical streams can be regarded as objects of a class which provides the public functions to open a stream, get a token from a stream, and close a stream. Parameters to the open function supply the details needed to define the types of tokens and their structures, giving flexibility for reuse with different languages.

Candidates for possible exceptions include :

- opening a non-existent stream
- invalid parameters with open
- using get on an unopened stream
- closing an unopened stream.

From knowledge of the domain of use, the action needed in these cases is known to be common to all anticipated uses of the class, so they are not treated as exceptions. The first three have error messages displayed and the program is terminated. The last one is ignored.

One situation which does use an exception is when get () finds an unrecognisable string. The languages differ in what they do when this situation occurs, so instead of trying to design a handler in the reusable class, we declare an exception and leave the provision of the handler to each compiler.

Note that we could have used exceptions for all of these candidates, which would have defined five exceptions. Having recognised four needed common treatment, a specific exception class defining the handlers for these four could have been designed. The instantiation using this specific exception class would leave the fifth exception uninstantiated, so that the provision of a handler could be made separately.

When we consider what action the handler of this exception might need to perform, it is clear that the public interface to the class will not provide sufficient details. In certain cases it will be required to report the position in the lexical stream at which the exception occurred. But this information is hidden in the lexical object. Following our proposal to declare a separate interface for exception classes, we declare the additional functions needed to retrieve this information in this interface. This limits what exception-handlers can access, gives them greater access than is available through the public interface, and avoids having more than is necessary in the public interface.

Figure 2 shows the outline structure of the reusable lexical class with three sections namely, public, exception, private. The exception called Unrec is associated with the get function. The specific exception class SP contains the handler for this exception, and can access the function position.

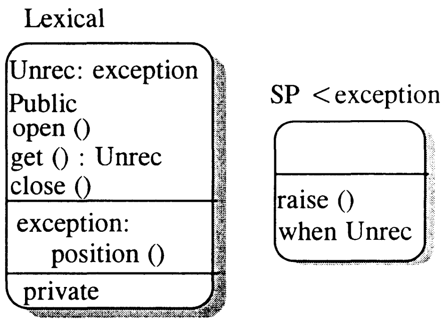

Figure 2 Reusable exception class

Tool support

The explicit declaration of exceptions at design time indicates clearly what possible exceptions have been identified for that reusable class. At the time of reuse, the designer must consider how these exceptions are to be handled, based on knowledge of the application domain. We propose a design tool to support designing for reuse with exceptions and reusing classes thus defined The tool will track the exception declarations and provide interactive information about the exceptions and the placement of their handlers.

Figure 3 shows a prototype model of a proposed tool for checking for exception class. This tool set can analyse existing design or design descriptions and provide advice and analysis for designing exceptions.

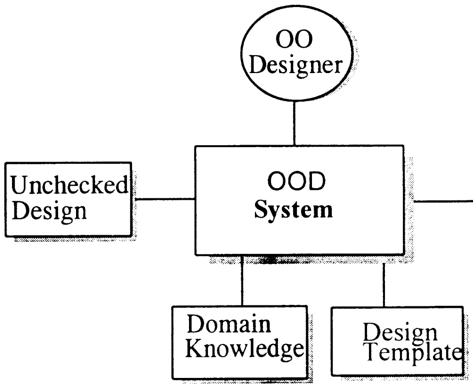

Figure 3 Object-oriented exception handling system

Conclusion

This paper discussed the issue of designing exceptions for reusable and reliable objects. As a result of initial investigations into the problems, we have proposed that exceptions can be treated as objects of a special superclass with handlers defined for them in specific exception classes. The need to have access to details of objects not normally available through the public interface has led us to propose a special interface for exceptions. We have outlined how this mechanism works for a small case study of a reusable lexical class. Tool support at the design level has also been discussed. Further work is planned to determine the effectiveness of these proposals on large case studies and to develop the proposed design support tool.

References

1. Coad, P. and Yourdon, E. *Object-Oriented Design*, Prentice-Hall, 1991.
2. Booch, G. *Object-Oriented Design and Application*, Benjamin-Cummings, 1991.
3. Meyer, B. *Object-oriented software construction*, Prentice-Hall, NJ, 1988.
4. Wirfs-Brock, R et al. *Designing OO software*, Prentice-Hall, NJ, 1990.
5. Rumbaugh, J et al. *OO modeling and design*, Prentice-Hall, NJ, 1991.
6. Sommerville, I. and Ramachandran, M. Reuse assessment, *First International workshop on Software Reuse*, Dortmund, Germany, 1991.
7. Ramachandran M and Taylor M. Towards a practical OOD, *BCS Information Systems SG conference*, Heriot-Watt University, Edinburgh, August 1993.

8. Ramachandran M. Reusable components for concurrent applications, *World Transputer Congress Workhop on Software Engineering for Parallel Systems*, Aachen, Germany, September 1993.
9. Dony, C. An OO exception handling system for an OO language, Gjessing and Nygaard (Eds) *ECOOP88* (European Conf. on OOP), Springer-Verlag, 1988.
10. Poigne, A. Partial algebras, suborting and dependent types - Pre-requisites for error handling in algebraic specifications, Gjessing and Nygaard (Eds) *ECOOP88* (European Conf. on OOP), Springer-Verlag, 1988.
11. Goodenough, J. B. Exception handling issues and a proposed notation, *Comms. of the ACM* 18(12), 1975.
12. Issarny, V. An exception-handling mechanism for parallel OO programming, *JOOP*, October 1993.
13. Stevens, A. C++ exception handling, *Dr Dobbs Journal* Vol. 18, PP. 393-401, 1993.
14. Cui, Q and Gannon, J. Data-oriented exception handling, *IEEE Trans. on software engineering*, Vol.18, No. 5, 1992.
15. Ichisugi, Y and Yonezawa, A. Exception handling and real-time features in an OO concurrent language. *Lecture notes in computer science*, vol. 491, 1991.
16. Gehani H. Exceptional-C or C with exceptions, *software practice and experience*, vol. 22, No. 10, 1992.
17. Oberweis, A and Stucky W. Exception handling in software systems - A literature survey, vol. 33, no. 6, *Wirtschaftsinformatik*, 1991.
18. Chen, Z. User responsibility and exception handling in decision support systems, *Decision Support Systems*, vol.8, no.6, 1992.