An Idiom for Exception Treatment in C++ and Java

José de Oliveira Guimarães
Departamento de Computação - UFSCar
São Carlos - SP, Brazil
jose@dc.ufscar.br

August 9, 2002

Abstract

The exception systems of C++ and Java use catch clauses for exception treatment. These clauses are statically attached to try blocks and cannot be reused. We propose an idiom that encapsulates catch clauses into methods of special classes thus giving the benefits of object-oriented programming to exception treatment. The result is an easy-to-use idiom that allows for code reuse and helps to enforce consistence among all exception treatments throughout the program.

1 Introduction

Languages C++ and Java employ similar exception systems. The main difference is that Java enforces the declaration of checked exceptions a method may throw.[1] Then we will use Java in the examples of this articles since its exception system is a bit more complete than that of C++.

Figure 1 shows an example of use of exceptions in Java. Inside a `try` block there may be `throw` commands, as in the example, or method calls that execute `throw` commands. Upon the execution of a `throw` statement, the control is transferred to the appropriate catch clause following the `try` block. Then, if statement

```
throw new InvalidTriExc(a, b, c)
```

is executed, there will be a jump to clause

```
catch ( InvalidTriExc e )
```

which acts like a procedure. The object thrown with `throw` is passed as a parameter to the `catch` clause. After executing the catch clause, the program continue with the statement following the last clause.

Exceptions in Java are objects of classes organized in a hierarchy. For example, class `TriangleExc` could be superclass of both `InvalidTriExc` and `NegSideExc`. Class `DivisionByZeroExc` can be subclass of `ArithmeticExc`. This exception organization, proposed by Dony [2], has been transformed into the design pattern Exception [7].

[1] That is, if a method may throw an exception, the exception should be declared in the method header: `void read() throws ReadException ...`
An unchecked exception need not to be declared. The class of an unchecked exception is always subclass of Runtime or Error.

```
...
try {
  if ( a < 0 ) throw new NegSideExc(a);
  ...
  if ( a >= b + c )
    throw new InvalidTriExc(a, b, c);
  ...
  t = new Triangle(a, b, c);
  }
catch( NegSideExc e ) {
  // treatment
  }
catch( InvalidTriExc e ) {
  // treatment
  }
```

Figure 1: Exception handling in Java

A catch clause

```
catch( TriangleExc e ) { ... }
```

can catch exceptions of class `TriangleExc` and its subclasses. Assuming `NegSideExc` is subclass of `TriangleExc`, the first catch of code

```
try {
  ...
  }
catch( TriangleExc e ) { ... }
catch( NegSideExc  e ) { ... }
```

will catch all exceptions of `NegSideExc` thrown in the `try` block. The second catch clause will never be called. The search for an appropriate catch clause is made in the declaration order.

The C++/Java mechanism for exception handling has some important characteristics, detailed below.

- The code for exception treatment, inside the catch clauses, cannot be reused since each catch clause is attached to a specific `try` block.
- A catch clause for an exception such as `NegSideExc` should be written every time this exception may be thrown. Then the program can have dozens of catch clauses for `NegSideExc`, which makes it difficult to enforce a standard treatment for this exception.
- The exception treatment cannot easily change at runtime since the catch clauses are statically attached to the `try` block.

Our idiom, called Exception Treatment, tries to remedy these shortcomings. It is presented in the next section. Alternatives for the idiom implementation are exposed in Section 3.

```
class CatchTri {
  public void select( Exception e ) {
    if ( e instanceof NegSideExc )
      treat( (NegSideExc ) e );
    else if ( e instanceof InvalidTriExc )
      treat( (InvalidTriExc ) e );
    else
      throw new NonCaughtExc(e);
    }
  public void treat( NegSideExc e )
    { /* treatment */ }
  public void treat( InvalidExc e )
    { /* treatment */ }
}
```

Figure 2: A catch class for triangle exceptions

2 The Exception Treatment Idiom

We are going to show the idiom[2] by rewriting the example of Figure 1. The catch clauses are put in a *catch class* `CatchTri` shown in Figure 2. For each clause there is a method `treat` with the same parameter and body (the treatment). Method `select` is responsible for selecting the appropriate `treat` method based on its parameter runtime type. The expression "`e instanceof NegSideExc`" returns `true` if `e` is an object of class `NegSideExc` or one of its subclasses. There are two `treat` methods in the example. Each one is identified by its parameter type. Then the message send "`treat( (NegSideExc) e)`", in which `e` is cast to type `NegSideExc`, will call method "`treat( NegSideExc e)`".

The `try` block of Figure 1 should be changed to

```
aCatchTri = new CatchTri();
try {
  ...
  // the same as before
}
catch( Exception e ) {
  aCatchTri.select(e);
  }
```

All `try` blocks should obey this format when using this idiom. Object `aCatchTri` should be an object of a catch class which has a `select` method. This object is called the *catch object*. The responsibility of choosing the exception treatment is changed from the runtime system (example of Figure 1) to method `select` made by the programmer.

If method `select` receives as a parameter an exception object that is not of a subclass of `NegSideExc` or `InvalidTriExc`, it throws exception `NonCaughtExc`. This is an unchecked exception — the programmer is not required to catch it. Exception `NonCaughtExc` is then thrown when there is a runtime error — an unexpected exception

[2] An idiom is a design pattern specific to a programming language.

is thrown and not caught. In the code of Figure 1, that would result in a compile type error. Hence our idiom transforms some compile-time errors into runtime ones.

Combination of Exception Treatments

The commands inside a try block may throw exceptions treated by two or more catch clauses. For example, a `try` block may throw exceptions `NegSideExc` and `ReadExc`. The former is treated by `CatchTri`. The last, by class `CatchRead`. Hence, our scheme, with just one `catch` after the `try` block, will not work. The `try` block should be expanded to two blocks, one for each catch class:

```
try { // outer
  try {  // inner
    // original try block
  }
  catch( Exception e ) { // treats NegSideExc
    aCatchTri.select(e);
    }
}
catch( Exception e ) {  // treats ReadExc
  aCatchRead.select(e);
  }
```

In class `CatchRead` of `aCatchRead`, method `select` should begin with

```
     if ( e instanceof NonCaughtExc )
       e = ((NonCaughtExc ) e).getException();
```

Method `getException` retrieves the exception object stored in the `NonCaughtExc` object by a `NonCaughtExc` constructor. If exception `ReadExc` is thrown in the inner try block, it is caught by the inner catch clause and passed to `select`. This method stores the exception in a `NonCaughtExc` object thrown in another exception — see Figure 2. This exception is caught by the outer catch clause. Method `select` of object `aCatchRead` is called, which retrieves the original exception from the `NonCaughtExc` object. Then the appropriate `treat` method is called by `select`.

Changing Exception Treatment

There may be more than one treatment for an exception. They can be put in `treat` methods of different catch classes. Then the programmer may choose which one to use, a decision that may be taken even at runtime. In general, treatments for related errors will be put in a single catch class and its subclasses will provide alternatives for error treatment. As an example, suppose class `CatchTri` of Figure 2 is now defined as an abstract class with concrete method `select` and abstract methods `treat`. Subclass `CatchTriExtreme` of `CatchTri` overrides both `treat` methods in such a way both will print an error message in the standard output and terminate the program. Subclass `CatchTriNice` also overrides the `treat` method so they do nothing — the error is not considered important. Another subclass could print an error message in a window and so on. Language Green [4], which has constructs to support idiom Exception Treatment, offers other possibility: to correct the error. Then you could ask the user help to correct the value of a `Triangle` side, for

example. We chose not to add this feature to the Exception Treatment idiom to keep it simple. But it can be extended to support this functionality.

The catch object to be used in a `try` block can be supplied by an abstract factory object. An abstract factory provides an interface for creating families of related objects [3]. The abstract factory object can have methods `getCatchTri` and `getCatchFile` for returning objects with interfaces equal to `CatchTri` and `CatchFile`. They would be used as in the case

```
try {
  ...
  }
catch( Exception e ) {
  AbstractFactory.factoryObject.getCatchTri().select(e);
  }
```

`AbstractFactory` is a class, `factoryObject` is a static variable of this class, and `getCatchTri` returns a catch object.

By changing the object pointed to by `factoryObject`, we change the exception treatment. If the abstract factory is used in all catch clauses of the program, all exception treatments are changed.

The Exception Treatment idiom was based on the exception system of the Green language [4] [5]. In Green, there is no catch clauses. An exception object attached to a `try` block is responsible to treat the exceptions the block may throw.

Applicability

The Exception Treatment idiom should be used when there are a lot of identical treatments for an exception in different places of the code. The treatment can then be coded in a single place, a `treat` method of a catch class. That helps the program maintenance since changes in a single `treat` method may affect exception treatment in all the code.

This idiom should also be used when the exception treatment should vary at runtime. By changing the catch object[3] at runtime, we change the `treat` methods that may be called, changing the exception treatment.

Structure

The structure of a catch class is shown in Figure 3. Method `select` calls the appropriate `treat` method according to the class of its parameter. A `try` block should follow the model below.

```
aCatchObj = new ConcreteCatch();
try {
  ...
}
catch( Exception e ) {
```

[3]Remember a catch object is an object of a catch clause. In the first example of this section, it is referred to by variable `aCatchTri`.

```
public class ConcreteCatch {
  public void select( Exception e )  {
    if ( e instanceof NonCaughtExc )
      e = ( (NonCaughtExc ) e).getException();
    // select a treat method based on e class OR
    // throw exception NonCaughtExc
    ...
  }
  public void treat(  Exception1 e ) { ... }
  public void treat(  Exception2 e ) { ... }
}
```

Figure 3: The structure of a catch class

```
  aCatchObj.select(e);
  }
```

Consequences

Exception treatment is reused because it is put in methods `treat` of catch classes. One may even subclass a catch class and overrides a `treat` method, changing then part of the exception treatment.

Using the Java exception handling system, one may use a catch clause for exception `TriangleExc` in a hundred places. But it will not be necessary one hundred different treatments. Probably just two or three different treatments are enough. Then there will be a lot of redundancy in the catch clauses, making maintenance hard. If one catch clause needs to be changed, probably all clauses similar to it should be changed too.

Our idiom puts a treatment for one exception in just one place — a method `treat`. Changing this method may change the treatment of an exception in dozens of situations.

3 Implementation

Besides being implemented directly by the programmer, method `select` may also be implemented using :

1. a software tool that generates it automatically based on user input, probably using a GUI;

2. an introspective reflection library or;

3. a compile-time metaobject protocol (MOP).

Option 1 is reasonably clear and will not be discussed in this paper. Option 2 demands all catch classes inherit from a class `Catch` with a single method, `select`. This method is

implemented using the Introspective Reflection Library (IRL).[4] All catch classes should inherit from `Catch` and define `treat` methods. When an object of a catch class receives a message "`select(e)`", method `select` of `Catch` is called. It searches and calls a `treat` method defined in the class of the object, which is a subclass of `Catch`. Method `select` knows which is the class of its parameter `e` through method `getClass()` defined for all objects. Using the IRL, `select` searches for a method called "`treat`" in the current object, `this`, and tests if the method found accepts `e` as parameter. If the `treat` method found does, `select` calls it.

There is a shortcoming in using the IRL for selecting a `treat` method. The IRL does not consider the order in which the `treat` methods are declared. Then if there are methods

```
void treat( TriangleExc )
void treat( NegSideExc  )
```

in which `TriangleExc` is superclass of `NegSideExc`, method `select` of `Catch` always selects the first method. Even when the exception thrown, parameter `e`, points to an object of `NegSideExc`. In this case, it would be more reasonable to use the second method. It is legal to choose the first method because it can accept a `NegSideExc` object as parameter.

Then the use of IRL to select a `treat` method should not be used when there is a subtype relationship among parameter types of `treat` methods.

A compile-time metaobject protocol (MOP) such as that of `OpenJava` [6] can be used to generate method `select`. Class `CatchTri` would be declared as

```
public class CatchTri
       instantiates SelectException {
  // as before
}
```

Class `SelectException` is called by the MOP, at compile time, to change class `CatchTri`. The only change it will do is to add to `CatchTri` a method `select` equal to method `select` of class `CatchTri` of Figure 2. `SelectException` asks questions to the compiler such as "which are the `treat` methods of `CatchTri` ?" and "what is the parameter type of this `treat` method ?". Then `SelectException` can easily generate a `select` method for `CatchTri`.

4 Conclusion

The Exception Treatment idiom widens the interactions between object-oriented programming and error treatment: catch clauses are encapsulated in methods of catch classes, catch classes may be inherited by other catch classes, exception treatment may be changed at runtime by using other catch objects, and a design pattern, abstract factory, may be employed to select a catch object.

The idiom foster code reuse because an exception treatment is written just once and put in a `treat` method. This also keeps the program maintenance simple: to change

[4]The IRL of Java is called Java Core Reflection. With it, one can know the class of an object at runtime, the methods of this class, the parameter types of each method, and so on. We can even call a method selected dynamically by a search made using the method name.

certain error treatment, one need to change just one `treat` method. If the idiom is not used, all catch's that treat that error should be found and changed.

The idiom has its drawbacks. It demands the creation of a catch object for each block, although one may use a static class variable thus saving an object creation. The idiom requires a `select` method which may be prone to error. It causes a runtime error when an exception thrown inside a `try` block is not expected by the `select` method of the catch object. This error would be pointed at compile time if the Java exception system were used. Then the idiom causes runtime errors in situations in which Java would point the problem at compile time. But the error is always signalled.

Idioms and patterns are only useful when they can be applied to a variety of contexts by different programmers. The Exception Treatment idiom lacks this practical test. Then we would thank reports of people that have used it, so we could add the conclusions taken from practice to a future article. In particular, some questions are important: is that common to throw exception `NonCaughtExc`[5] ? Is it really useful to subclass catch classes ? How deep are the catch-class hierarchies ? When using the idiom, is the number of `treat` methods much smaller then the number of catch clauses when not using it ? Is the `select` method prone to error and difficult to maintain ? Does the `select` method implemented using the Introspective Reflection Library work well ? It does not if there is a subtype relationship among parameter types of `treat` methods.

References

[1] Coplin, J.O.; Schmidt, D.C., eds. *Pattern Languages of Program Design 1*, Addison-Wesley, 1995.

[2] Dony, C. An Object-Oriented Exception Handling System for an Object-Oriented Language. *Lecture Notes in Computer Science*, Vol. 322, *ECOOP 88*.

[3] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[4] Guimarães, José de Oliveira. The Green Language. Available at http://www.dc.ufscar.br/~jose/green/green.htm.

[5] Guimarães, José de Oliveira. The Green Language Exception System. Available at http://www.dc.ufscar.br/~jose/green/green.htm.

[6] Tatsubori, M. An Extension Mechanism for the Java Language. Master Thesis, University of Tsukuba, 1999.

[7] Wolf, Kirk and Liu, Chamond. New Clients with Old Servers: A Pattern Language for Client/Server Frameworks. In [1].

[5] That would indicate an exception was not caught by the catch object.