The Green Language Exception System

José de Oliveira Guimarães
Departamento de Computação - UFSCar
São Carlos - SP, Brazil
13565-905
Phone: 55 16 260-8607
Fax: 55 16 260-8233
jose@dc.ufscar.br

Abstract

Exception systems have become popular because they uncouple error detection and handling thus allowing us to produce better software. However, the exception handling systems of many object-oriented languages benefit very little from object-oriented programming. The catch clauses following a try block are isolated pieces of code: they do not belong to classes and cannot be reused. This paper presents the object-oriented exception system of the Green language, which fully uses the object-oriented programming features. That includes the use of classes, types and subtyping, objects, and inheritance. The result is a powerful exception system completely integrated with the language.

1 Introduction

Exception systems are language features used to intercept the control flow when some pre-defined assertions are violated. An assertion may be defined by the language or the programmer. "array index out of bounds" and "division by zero" are examples of assertions defined by the language. A programmer may define her own conditions that results in exceptions. The assertion that triggers an exception may detect an error or simply an abnormal state.

When an assertion fails, an exception is signalled, thrown or raised. The program control is transferred to the exception handler which is a piece of code able to handle this extraordinary state. As an example, the following code shows an example of exceptions in Java.

```
try {
  if ( a >= b + c )
    throw new TriangleExc(a, b, c);
  t = new Triangle(a, b, c);
  ...
} // end of try
catch( TriangleExc e )
{
  System.out.println("Triangle is" +
    " not valid");
}
```

Keyword `throw` is used to *signal*, *throw*, or *raise* an exception. After its execution the control is transferred to the *catch clause* after the closing `}` of the `try` statement. This *clause* acts like a procedure and is called the *exception handler*. The method that signaled the exception (the one to which belong the code above) is called the *signaler*. The object of class `TriangleExc` created in the `throw` statement is an exception object. It is passed as parameter to the *catch clause*. This object is used to pass information on the error or exceptional condition to the handler. The exception classes like `TriangleExc` may be organized in a class hierarchy, an idea first proposed by Dony [1]. For example, `TriangleExc` may inherit from `PolygonExc`.

The `throw` statement could be in a method called inside the try block:

```
void getText()
{
  try {
    ...
      /* open can throw
         exception OpenFileExc */
    file.open(name);
    ...
  }
  catch( OpenFileExc e )
  {
    ...  // catch body
```

```
  }
} // end getText
```

An exception can be raised in method `open` and handled outside this method. It is supposed that the caller of `open`, the code shown above, is better equipped to handle the exception than method `open`. Then exception systems uncouple the detection of an abnormal condition, the exception, from its handling. One is in `open` and the other is in `getText`.

In Java [2] and C++ [3], after an exception is thrown, the signaler is terminated and the execution continues in a catch clause. This is called the "termination" model of exception handling. Other models of exception handling will be discussed in Section 3.

The Java example just given show us some characteristics of the Java/C++ exception systems, detailed below.

- It is only partially object-oriented. The exceptions are objects passed as parameters in `throw` statements, thus allowing communication between the signaler and the handler. But the catch clauses are isolated pieces of code attached to `try` blocks. They do not belong to classes and cannot be reused.[1] Each time an exception, say `OpenFileExc`, must be caught, a catch clause should be written. A standard handling for this exception throughout a program is then difficult to enforce.

- As a consequence of the previous item, catch clauses are a procedural sea in an object-oriented ocean. The exception signaling and handling need *special rules* of correctness, as to declare the exceptions a method may throw after its header:

  ```
  void open(name : String)
          throws OpenFileExc
  ```

  Inside this method, an instruction "`throw exp`" is only legal if the type of `expr` is subtype of `OpenFileExc`, another special rule.

- Maintenance is hard, since an exception is handled in several catch clauses (scattered throughout the program) that are generally very similar or equal to each other. If one of them is changed, in general the others will need to be changed too.

- The exception handling is made in the catch clauses that follow the try block. So, the signalling point, the try block, is coupled to the handling, the catch clauses. Then, the catch clauses are fixed statically in the same way a procedure call is bound statically to a procedure in a procedural language. Polymorphism is not possible; that is, to dynamically change the catch clauses attached to a `try` block is not possible. It will be seen later in this paper why this is important.

This article presents the Green language exception handling system (EHS), which objectfies as much as possible the Java/C++ EHS. We wanted a C++/Java-like model but with the advantages of object-oriented programming. Before introducing the EHS of Green, we will show a little bit of this language. Green [4] [5] is an object-oriented, statically-typed, and reflective language which supports runtime metaobjects, introspective reflection, garbage collection, parameterized classes, and classes as first-class objects. As an example of Green syntax, Figure 1 shows a class `Store` with methods `set` and `get`. In general, the language follows a Pascal-like syntax with a bit of C.

In Green, there are separate hierarchies for subclassing and subtyping. A class is *subclass* of another if it inherits from it. The type of a class is a set with the signatures of the public class methods. A signature of a method is composed by its name, parameter types and return value type (if it exists). As an example, the type of class `Store` of the figure is

```
{ set(integer),
  get() : integer, ... }
```

in which `{` and `}` are used to delimit the set. The "`...`" are methods inherited from `Any`, the top-level class inherited by every other class. A class S is *subtype* of class class T if S has at least all method signatures of T. It does not matter whether S inherits from T or not.

A method of a *subtype* should have the same argument types and return value type as the supertype method with the same name. The covariant and contravariant rules [6] do not apply.

This paper is organized as follows. Section 2 introduces the Green exception system. Other exception handling models are explained in Section 3. Section 4 makes general comments on the Green ex-

[1] However, a catch clause can call a procedure to handle the exception. And this procedure can be reused. But then we introduce one more level linking the exception to the handling. Anyway, a non-object-oriented reuse is obtained.

```
class Store
  public:
      /* this is a comment.
         Methods begin with
         keyword proc */
    proc set( pn : integer )
      begin
      if pn >= 0
      then
        n = pn;
      endif
      end
    proc get() : integer
      begin
      return n;
      end
  private:
    var n : integer;
end
```

Figure 1: Class `Store`

ception system and compares it to other language systems. The implementation of exception handling is shown in Section 5. Section 6 concludes and presents some comments on future works.

2 An Object-Oriented Exception System

This section explains the Green exception system by gradually unfolding its syntax, semantics, and relationships to other features of the language.

2.1 A Light Introduction

The main idea behind the Green exception system is to transform the *catch clauses* of Java/C++ into methods of a *catch object*. Our goal in the design of the Green Exception Handling System was to add more object-oriented programming to the Java/C++ model. The example in Green of the first Java code shown in Section 1 is:

```
  // comment
  // creates a new object
catchTri = CatchTriangleExc.new();
  // the try block always
  // has a parameter
try(catchTri)
  ...
  if a >= b + c
  then
    exception.throw(
      TriangleExc.new(a, b, c) );
  endif
  t = Triangle.new(a, b, c);
  ...
end // end of the try block
```

The try block begins with `try` and ends with `end`. The `try` keyword takes an object as "parameter". This object (or expression) should belong to a user-defined *catch class*. `CatchTriangleExc` used in this example is a *catch class*. It is shown in Figure 2. `TriangleExc` is an exception class as before and may belong to an exception hierarchy.

The code of the Java *catch clauses* is put in `throw` methods of a *catch class*. A catch class as `CatchTriangleExc` is a regular class except that some of its methods should have the name "`throw`". In Green a class may have two or more methods with the same name if the parameter types or number of parameters is different. Then there may be several `throw` methods in a class.

A *catch class* need not to be subtype or subclass of any class. It only needs to have at least one `throw` method.

Inside the try block in the example shown above, an exception is thrown by sending message `throw` to variable `exception`, which is a language reserved word. Then method `throw` of object `catchTri` is executed. That is graphically illustrated in Figure 3. When message `throw` is sent to `exception`, a search is made in the `catchTri` object for a method `throw` that can accept a `TriangleExc` object as parameter. This method is found and executed. The execution resumes at the statement following the `try-end` block. In fact the search is made in `CatchTriangleExc`, the class of `catchTri` which has the code corresponding to the catch clauses of Java/C++. Note the search for a method `throw`

```
class CatchTriangleExc
  public:
    proc throw( exc : TriangleExc )
      begin
      Out.writeln(
           "Triangle is not valid");
      end
end
```

Figure 2: A catch class

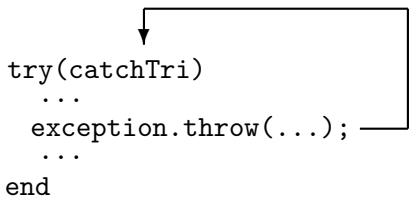


Figure 3: Graphical representation of exception handling

is very similar to the search for a catch clause in Java/C++.

Everything happens as if `exception` referred to the same object as `catchTri`. It is as if the call

```
catchTri.throw(
    TriangleExc.new(a, b, c);
```

were made. After that, the execution continues after the try block.

In this example, if the exception `TriangleExc` is signalled, method `throw(exc : TriangleExc)` of class `CatchTriangleExc` is called. This means the `throw` method cannot access the local state of the try block — in the Java example, the catch clause could change variables `a`, `b`, and `c`. Note that method `throw` has read-only access to these variables which are kept in the parameter of class `TriangleExc` passed to the method.

Inside the try block the `exception` object is *enhanced* by the methods of `catchTri`, thus allowing it to receive `throw` messages. Assume `exception` is a special variable declared somewhere else. We will soon explain that.

After a try block there may appear a `finally` block. The code inside this block is executed whether or not an exception is thrown inside the associated try block. The `finally` block was not transformed into a method of a catch class because it generally does clean-up actions that depend on local and instance variables. These variables, of course, cannot be accessed in a method of a catch class. It seems that finally blocks almost always need to access the local state. This is not too necessary in catch clauses (the handlers). So we chose to transform only the handlers into methods.

Exceptions in Green are organized in a hierarchy with class `Exception` at the root. Every exception class like `TriangleExc` **must** be subclass of class `Exception` (directly or indirectly). There are some pre-defined exception classes such as `DivisionByZeroException`, `MessageSendToNilException`, and `PackedException`.

```
proc compile( name : String )
       ( exception : CatchCompiler )
  begin
    // 1
  try( CatchFileExc.new() )
    ... // 2
      // declare variable f
    var f : File;
    f = File.new();
    f.open(name);
    try( CatchSyntaxError.new() )
      ... // 3
      text.parse();
      ...
    end // end try
  end // end try
  end // end proc
```

Figure 4: Example of use of types in the exception system

2.2 Typing Exceptions

Green uses the type system to check the correctness of the signalling and handling of exceptions. The details will be shown using the example of Figure 4. Method `compile` compiles a file named `name` passed as parameter. It opens this file and calls other methods to parse, build the abstract syntax tree, etc.

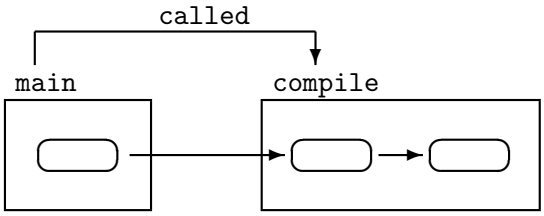
In the header of method `compile`, after the declaration of parameter `name`, there appear the declaration of the special variable `exception`. This variable can only be declared in this place — it cannot be a local or instance variable. Then `exception` is a kind of special method parameter through which typing is introduced to assure the correctness of the use of exceptions.

In Figure 4 the type of `exception` at `// 1` is `CatchCompiler`, which is its declared type. Assume this class has a single method,

```
throw(GeneralExc)
```

`GeneralExc` is the type of the single parameter.[2] Then at `// 1` only exception `GeneralExc` or its subtypes can be thrown. This is because an exception signaling would be "`exception.throw(expr)`" which is correct only if the type of `expr` is subtype of `GeneralExc`.

[2] In fact this is the method signature, not the method.



Great rectangles: methods
Rounded rectangles: catch objects
The arrows link the objects of the stack of catch objects

Figure 5: Representation of a stack of catch objects

At point `// 1`, it is legal to throw an exception if it can be caught by an object of `CatchCompiler`. This is checked by the compiler. Then, the type of `exception` is `CatchCompiler` at that point. Point `// 2` is inside a `try` block with parameter `CatchFileExc`. Then at `// 2` it is legal to throw an exception if it can be caught by a `CatchFileExc` or a `CatchCompiler` object. A statement "`exception.throw(expr)`" at point `// 2` would trigger at runtime a search for a `throw` method that can accept `expr` as parameter. The search would start at *catch object* "`CatchFileExc.new()`" and, if necessary, continue at the object `exception` passed as parameter. Remember `exception` is a special parameter whose type is `CatchCompiler`. The type of `exception` at point `// 2` is the union of types of objects "`CatchFileExc.new()`" and formal parameter `exception`:

```
type(CatchCompiler) ∪
type(CatchFileExc)
```

We use `type(C)` for the set of method signatures of class `C`, its type. A class `S` is subtype of class `T` if `type(T)` $\subset$ `type(S)`. The type of `exception` is statically enhanced inside the `try-end` block by the type of the `try` parameter.

At `// 3`, an exception signalling is legal if it can be caught by an object of `CatchSyntaxError`, `CatchFileExc`, or `CatchCompiler`. At this point, the active catch objects and methods are shown in Figure 5. Assume `main` is the method calling `compile`. The leftmost rounded rectangle, inside `main`, is a catch object whose class is subtype of `CatchCompiler`. The catch object in the middle is a `CatchFileExc` object corresponding to the first `try-end` command of method `compile`. The rightmost object corresponds to the inner `try-end` command, a `CatchSyntaxError` catch object. When an exception is thrown at `// 3`, the search for a suitable `throw` method is performed from the rightmost to the leftmost catch object. Assuming they compose a *stack of catch objects*, the search is performed in a top-down fashion. The pushing and popping of objects in this stack is made by `try-end` commands. At the start of a `try(obj) ... end` command, `obj` is pushed into the stack. After `end`, `obj` is popped off the stack.

The type of `exception` at `// 3` is

```
type(CatchCompiler) ∪
type(CatchFileExc) ∪
type(CatchSyntaxError)
```

When an exception is signalled, as in "`exception.throw(exc)`", the run-time system scans the *stack of catch objects* from top to bottom looking for a `throw` method that accepts `exc` as parameters (from right to left in Figure 5). First it examines the top catch object and looks for a method `throw` that accepts `exc`. This search begins at the first textually declared `throw` method in the catch-object class and continues downwards. Then, the order in which the `throw` methods are textually declared in the catch-object class is important. The *search* mechanism of this exception system is similar to the mechanism of Java/C++ — the former looks for a `throw` method in a stack of catch objects and the last looks for a catch clause in a stack of nested try statements.[3]

There is another observation on the search for a `throw` method in a catch object. The class of the catch object, say `B`, may be *subclass* of a class `A`. In this case, the run-time system will look for a appropriate `throw` method in `B` and then in `A`. In both cases the search begins at the first textually declared method as usual.

A call "`f.open(name)`" to a method like `open`, whose header is

```
proc open( name : String )
    ( exception : CatchOpenFile )
```

is translated automatically by the compiler into

```
f.open(name)
       (exception)
```

This call is type checked like a regular call. There are two parameters: `name` and `exception`. In both cases, the real-argument type should be subtype

[3]It is as if every try statement had references to its catch clauses and every try were pushed into the stack by the run-time system before its first statement and popped off after its last statement.

of the formal-argument type. Let us use `type(C)` for the type of class `C` or parameter/variable `C`. Then `type(exception)` should be subtype of `type(CatchOpenFile)` in which `exception` is the real argument and `CatchOpenFile` is the type of the formal parameter `exception` in the declaration of method `open`. Remember this `exception` object, passed as parameter to `open`, is either declared in the header of the current method or it is brought to life by a `try` block which contains the call to `open`.

Java requires a similar test, which is semantically equivalent to the rule just described: if a statement of a method can throw a checked exception, the method should catch it or the exception name should appear in the section `throws` at the method header.

The Green language demands that the declared type of `exception` has `throw` method signatures for the exceptions that may be signalled inside the method (as `compile`) but that are not caught by the `try-end` blocks.

So, if there were no try-end statement in method `compile` of Figure 4, class `CatchCompiler` (the type of `exception`) should have all the methods `throw` of `CatchFileExc` and `CatchSyntaxError`. Only in this way no compile-time error would be signalled.

Suppose method `compile` of Figure 4 belongs to class `Compiler`. If a subclass of `Compiler` overrides `compile`, it must keep the *type* of `exception`, `CatchCompiler`. That could be different. One could use a supertype of `CatchCompiler` as type of `exception` in the overridden subclass method. That would mean the subclass method throws a subset of those exceptions method `compile` of `Compiler` throws. Although this contravariant rule is more flexible, we chose to forbid any redefinition of parameter types in subclasses to keep the type system simple.

2.3 Catch Objects

In Green one may declare a classless object as in prototype-based languages. These objects may be used as catch objects. The language provides classless objects `CatchAll` and `HCatchAll` that catch every exception and print a message in the standard output. Besides that, `HCatchAll` will also terminate the program. The use of these objects make it easy to program when no sophisticated error handling is necessary.

A classless object in Green is called a "class object", which is in fact an object that represents a class. A class object plays the role of a metaclass. It is as if we gathered the static methods and variables of a Java/C++ class and had put them in a classless object. As an example, `CatchAll` is declared as

```
object CatchAll
  public:
    proc throw( exc : Exception )
      begin
      set(exc, exc.getClassObject());
      end
    ...
end
```

Since `CatchAll` is already an object (and not a class), we can use it directly in a try statement:

```
try(CatchAll) ... end
```

It is not necessary to create a new catch object each time this try statement is executed (as in `try(C.new()) ... end`).

A class object can receive a message as a normal object. For example, class object `Runtime` of the Green Library has a method `exit` which may be called as "`Runtime.exit(1)`". The programmer may declare both a class object `A` and a class `A` in a file called "`A.g`". They are never confounded.

An unchecked exception in Green is an exception that need not to be caught by the user code. Then a method can throw an unchecked exception even without declaring an `exception` parameter. However, unchecked exceptions in Green can only be thrown by the runtime system, never by the user code directly. The runtime system may throw the unchecked exceptions "stack overflow", "array index out of bounds", "division by zero", etc. Then there should exist `throw` methods in the source code corresponding to these exceptions. These methods would be, in other languages, the default handlers for these unchecked exceptions. In fact, before a Green program starts its execution, a *catch object* of class `HCatchUncheckedException` is pushed into the *stack of catch objects*. This class has a `throw` method for each unchecked exception. Each method prints an error message and terminates the program. If the user wants to do something else, she can override this catch object at runtime through class object `Runtime`:

```
Runtime.setCatchUnchecked(
    MyDefaultHandlers.new() );
```

In this example, the default handlers of the unchecked exceptions will be first searched for in `MyDefaultHandlers`. If the appropriate han-

dler is not found in this class, the search continues at class `HCatchUncheckedException`. Class `MyDefaultHandlers` could define, for example, method

```
throw(e : OutOfMemoryException)
```

to handle all errors caused by lack of free memory.

Every exception signalled by the programmers' code is a checked exception and should be handled somewhere — the compiler prohibits that the programmer' code signals an exception that will not be handled at runtime.

3 Other Exception Handling Models

After an exception is thrown in Green, the program execution does not continue in the statement following the one that signalled the exception. Therefor we say Green follows the *termination model* of exceptions: the current line of execution is *terminated* and control is passed to same other place.

In the resumption model [7], the execution continues in the statement following the exception signalling.

The termination model has a variation called the retry model. After the exception is thrown in a try block, it is handled and then the execution continues at the beginning of the try block. The replacement model [8] allows the handler of an exception to supply a value which was expected to be returned by the signaller of the exception.

Of course, there are endless variations of these models and names given to them. We will discuss some of then below. A complete discussion of all models is outside the scope of this paper.

Beta [9] [10] supports two models of exception handling. The first one is a static model: it does not depend on the calling stack which handler is called when an exception is raised. The second model is similar to the Java/C++ mechanism, although implemented using constructs specific to Beta like virtual patterns.

Language Eiffel [11] uses the retry model and requires the use of the `retry` command to start again a method that is handling an exception, which corresponds to a `try-end` block in Green. If this command is not used, another exception is raised indicating the method has failed.

In Lore [1], exception handlers can be attached to statements, classes, and exception classes. When an exception is signalled, the search for a handler is first made in the handlers dynamically nested on the signalling statement, which is what is made in Java/C++. If no handler is found, the search continues in the class of the signaller and its superclasses — handlers may be attached to classes. If no handler is found, the handler attached to the thrown exception is used. If there is no handler in the exception class, the exception is propagated to the caller.

In Smalltalk, a block may be guarded by several handling blocks [12]:

```
[ ... MyException signal ... ]
  on: MyException do:
    [:anException |
       ... handle it ... return]
  on: MyException2 do:
    [:anException |
       ... handle it ... resume]
  on: MyException3 do:
    [:anException |
       ... handle it ... retry]
  ensure: [ ... clean-up ... ]
```

The exception is signalled by "`MyException signal`" and caught by one of the blocks following the `do:`. The block following `ensure:` is always executed.

Another way to signal an exception in Smalltalk is to send a message `error:` to `self`. A class method will be called to handle the exception. There is no search in the stack of called methods for a handler — exceptions are never propagated to callers [13] [14].

Mitchell et al. [15] describes a model for exception handling that uses reflection. Handlers are put at the metalevel and can be reused and replaced (by replacing the metaobject they are attached to). An exception thrown at the base level is intercepted by the metalevel which chooses the appropriate handle.

There have been proposals [13] [16] [17] [18] [19] [20] of other exception system models. They compose the models above or introduce other relationships between the signaler, the exception, and the handler. These proposals are not directly related to the exception system described in this paper and, for lack of space, no detailed discussion of them will be made. The interested reader should consult Garcia et. al [7].

4 Discussion

In Green, the use of exceptions is checked by the type system. There are three situations that need to be checked:

1. in a statement "`exception.throw(expr)`", `exception` should have a method `throw` that can accept `expr` as parameter. This means that this exception will certainly be handled either by a try block of the current method or by the caller of the current method. Remember that the type of `exception` is enhanced inside `try-end` blocks;

2. suppose `open` is declared as
   ```
   proc open( name : String )
       (exception : CatchOpenFile)
   ```
   In a call "`file.open(name)`", the type of variable `exception` of the caller, passed implicitly as parameter to `open`, should be a subtype of `CatchOpenFile`. Remember this call is in fact "`file.open(name)(exception)`" since we assume `exception` is being passed as parameter. This means the caller of `open` knows how to handle the exceptions `open` may signal;

3. a method should "export" all exceptions it may throw and does not catch. In method `open` of the previous item, formal parameter `exception` should have a type with a `throw` method for each exception that may be thrown and is not caught inside method `open`. By declaring parameter `exception`, method `open` is allowed to throw exceptions it cannot handle. If `exception` is not declared, this method must catch all exceptions its code may throw.

Miller and Tripathi [8] reported some mismatches between object-oriented programming and exception systems. They divided these mismatches in four groups which are related to abstraction, encapsulation, modularity, and inheritance. The problems related to abstraction, encapsulation, and inheritance are either due to the very nature of exception systems (an exception reveals something on the code that throws it) or to bad programming practices. So, let us concentrate on modularity problems.

Modularity is a property that makes a piece of code (module/class/method) easy to be changed without invalidating other modules or classes that use it. Exception handling systems of Java/C++ are harmful to modularity because changes in the exception class hierarchy or in methods that signal exceptions *may* require changes in the handling of the exceptions, which are spread throughout the code in catch clauses that follow `try` statements. As an example, cited by Miller and Tripathi [8], method `open` of class `Stream` can throw exception `InputErr`. This method is called in the following code.

```
  // Java code
try {
  aStream.open();
  ...
}
catch( InputErr e )
{ ... }
```

Method `open` may be modified to throw one more exception, `StreamOpened`, which is subclass of `InputErr`. If this new exception is thrown in the call to `open` in the above code, the catch clause for `InputErr` will be used, which is at least non-adequate. It can be worse. Method `open` may be changed to throw another exception not related to `InputErr`, causing compile errors almost anywhere this method is used.

Both problems are easily solved in Green. A class `Stream` should be delivered with a class `CatchStreamExc` to handle the exceptions it may throw. For each exception the methods of `Stream` may throw, there should be a `throw` method in `CatchStreamExc`. This class could have several subclasses thus offering several options for error handling.

The evolution of classes `Stream` and `CatchStreamExc` is tied. The addition of exceptions to `Stream` should be followed by the addition of `throw` methods to `CatchStreamExc`. Changes in the meaning of exceptions can also be made in both classes.

Of course, the programmer could define a class `MyCatchStreamExc` to handle the same exceptions that class `CatchStreamExc` handles. In this case, changes in `CatchStreamExc` should trigger changes in `MyCatchStreamExc` and vice-versa. Note that the `throw` methods of `CatchStreamExc` do not work like default handlers for the set of exceptions the code of `Stream` may signal. They are not default handlers because the programmer may choose either class `CatchStreamExc` or `MyCatchStreamExc` to handle the exceptions signalled by `Stream`.

The joined use of `Stream` and `CatchStreamExc` solves many instances of a problem we will call "the

evolution of method interfaces" (signatures) [21]. Suppose method `close` of `Stream` is declared as

```
proc close()
  (exception : CatchStreamExc)
```

A subclass `MyStream` of `Stream` redefines `close`, which may now throw an exception `E` not present at `CatchStreamExc`. This means the subclass method adds a new exception to its interface which does not appear in the superclass method interface. This is illegal in Java and should be illegal in any object-oriented language. In fact, in Green the `close` method of the subclass must have the same interface as in the superclass. This problem is solved in Green by adding a method

```
proc throw(e : E)
```

to `CatchStreamExc`. Then the language supports the evolution of method interfaces. However, Green does not solve this problem completely. In the previous example, there may be two classes `CatchStreamExc` and `MyCatch` with handlers for the exceptions `close` may throw:

```
try(MyCatch.new())
   ...
   file.close();
   ...
end
```

Then a method `throw(e : E)` should be added to `MyCatch` too, which is unexpected and can only be discovered by examining all the code.

Consider the following code:

```
try( CatchA.new() )
   try( CatchB.new() )
      ...
      exception.throw(E.new());
      ...
   end
end
```

`CatchA` defines a method `throw(e : E)`. Class `CatchB` does not define any methods capable of handling exception `E`. That means the exception thrown in the code above will be handled by method `throw(e : E)` of `CatchA`. Suppose method `throw(e : E)` is added to class `CatchB` — maybe it was added to correct the "evolution of method interfaces" problem which appeared elsewhere in the program.

Now the exception thrown in the code above will be handled by a method of `CatchB`, which was not the intention of the programmer when she/he wrote the code. The Green flexibility of grouping reusable handlers in classes also brings this inevitable problem to the language.

Polymorphism is one of the key features of object-oriented programming. Through it, a message send "`data.search(5)`" does not need to be linked to a particular method `search` at compile time. The binding message/method is delayed till runtime when the message is sent.

This same flexibility is brought to exception handling by the Green exception system. The *catch clauses* of Java/C++ are encapsulated in objects that are parameters of `try-end` blocks. By changing the parameter you can change exception handling. Of course, this should only be used to easily change the configuration of the software. It is easier to change the runtime class of the try parameter than to change catch classes, inheritances, and exception classes. This feature should not be used to change exception handling many times during runtime.

Lippert and Lopes [22] use an aspect-oriented programming extension to Java, AspectJ, to investigate the use of aspects [23] in exception handling. In AspectJ, one can separate the code for exception handling (catch clauses) and signaling (`throw` statements) from the program code. In Green, only the code for exception handling can be put in *catch classes*, separated from the code that throws exceptions. The code for detecting the exceptions, the `try-end` blocks, should be embedded in the source code.

The code for exception handling and signaling is put in aspect classes. An aspect class wrappers a class at compile time in a way similar to class views [24], a compile-time wrapper mechanism based on the proposal of Ossher and Harrison [25]. Each method of the aspect class, called *advice*, may wrap one or more methods of the class. An *advice* may check pre- and post-conditions, throwing exceptions if necessary. It may also catch exceptions thrown by the method it wraps. In this function, an advice works like a method of a catch class in Green.

AspectJ is both more general and more restrictive than Green concerning exception handling. It is more general because exception signaling can also be separated from normal code and because one can succinctly wrap an advice to a great number of methods. One can say "apply this advice to all methods returning an object, regardless of its name and parameter types".

AspectJ is more restrictive than Green because

- handlers are fixed at compile time. In Green, changing a `try` parameter at runtime (a catch

object) changes the exception handling;

- only methods may be wrapped. Then you cannot signal and handle an exception in a piece of code interior to a method as is made in a `try-end` block.

AspectJ and Green were designed with different objectives in mind. This can be clearly seen when we examine the advantages and disadvantages of both approaches. Some of the features of AspectJ seem to be outside the scope of a regular Exception Handling System, as to apply a wrapper to a great number of methods.

The number of different handlings for a given exception is usually small. If there are fifty *catch clauses* for an exception `OpenFileExc`, probably there will be a few different handlings in the catch clauses. This common sense observation was corroborated by *one* case study made by Lippert and Lopes [22]. That means catch clauses of Java/C++ introduce redundancy in programs making them more difficult to maintain: if one *catch clause* should be changed, probably all clauses similar to it should be changed too. Green *catch classes* and AspectJ aspect classes eliminate this redundancy by transforming catch clauses into methods.

The *catch clause* of the example in Java of page 1 could access the local and instance variables visible at that point. When this example is coded in Green, this *catch clause* is transformed into a `throw` method of a *catch class*. Obviously the `throw` method cannot access the local and instance variables anymore. This is a disadvantage of the Green exception system. However, we believe the access to local state by the handler is not generally necessary. The reason is the following: in Java/C++, a program may have a lot of *catch clauses* to handle an exception, say `TriangleExc`. These clauses are attached to several `try` blocks throughout the code. In general, the code inside these *catch clauses* will be equal to each other both syntactically and semantically. Since the clauses are spread in different methods/classes and are equal, they do not access local and instance variables which depend on method/class.

The exception handling in the `throw` methods may need to access the local/instance variables of the method throwing the exception. If this access is read-only, we can pass local/instance variables as parameters to the constructor of the exception object:

```
// filename is a local variable
exception.throw(
  OpenFileExc.new(filename) );
```

However this is not enough in two situations:

a) the local/instance variables should be modified in the exception handling (which would be the catch clauses of Java/C++ or `throw` methods of Green). In Java/C++ this is easy as the catch clauses are within the scope of local/instance variables. In Green, the `throw` methods do not have access to the local state of the method throwing the exception;

b) the handler (`throw` method) should access the local state and this access is different in different places the handler is used. For example, in one place the handler needs to close a file, in other it needs to show a message in the screen, and in another it needs to close a window and delete a file. Each case demands a different handler because variables of different types would be manipulated and different actions should be taken.

If the local/instance variables should be modified by the exception handling we should abandon the Green EHS. Or better, we should try to simulate the Java/C++ EHS using Green. This can be made using class object `Catch` as the catch object:

```
try(Catch)
   ...
end
  // Green employs a Pascal-like syntax
  // for case statements
case Catch.getClassException() of
   OneExc:
        // handler for exception OneExc
      ...
   TwoExc:
        // handler for exception TwoExc
      ...
end // case
```

Method `getClassException` of `Catch` returns the class of the exception thrown in the `try-end` block (or `nil` if none was thrown). Each of the case options is a class name and one option will be selected according to the class of the exception thrown in the `try-end` block. This is a rough simulation of the Java/C++ command

```
try { ... }
catch ( OneExc e ) { ... }
catch ( TwoExc e ) { ... }
```

`Catch` has a `throw(Exception)` method that does nothing.

5 Implementation

There is a Green compiler available at [5] which translates Green to Java. This compiler implements all the features described in this article but metaobjects.[4] The implementation of exception handling is very similar to the implementation of the idiom[5] Exception Treatment [26]. To each catch class the compiler adds a method `select` taking one parameter of class `Exception`. This method calls one of the `throw` methods based on the class of the parameter. In Green a try statement

```
try(catch)
   exception.throw(anException);
end
```

is roughly translated to the Java code

```
try {
   push catch into the stack
   throw anException;
} catch( Exception e ) {
   catch.select(e);
}
finally {
   remove top stack element
}
```

Method `select` selects the adequate method of object `catch` to handle the exception.

6 Conclusion

Green considers not only exceptions but also groups of *catch clauses* as objects, bringing the object-oriented advantages to its exception system. There are catch and exception objects, catch and exception hierarchies, redefinition of `throw` methods in subclasses, polymorphism, reuse of code for error handling, and use of the type system for checking exception signalling and handling.

A method with a `try-end` block assigns to itself the responsibility of handling at least some of the errors signalled inside it. But the error handling is really made in catch objects that can be replaced dynamically. Therefore, Green introduces polymorphism to the exception system and uncouples error detection (`try-end`) from error handling (catch objects).

The exception systems of Ada, C++, and Java have some points in common. They either allow the runtime error "Exception thrown and not handled" or require specific rules for exception signalling and exception declaration (in a method header) to prevent this error from happening. These ad-hoc rules made only for the exception system are largely absent in Green, which uses the type system for this purpose. Green has its specific rules, but they are object-oriented in nature.

As a side effect of using a parameter `exception` and catch objects for signalling and handling exceptions, we as language designers can understand better the workings of exception systems. As an example of that, we can cite the `try-end` blocks which, in Green, just enhance the type of parameter `exception`. And the search for a handler is modelled by a Design Pattern, Chain of Responsibility [27].

Catch classes can be organized in a hierarchy which can store information on how to handle abnormal situations or errors. A subclass may redefine a `throw` method that handles an error in a different way from the superclass.

Green offers an Introspective Reflection Library that allows us to examine the *stack of catch objects* at runtime, the methods of each object, their parameters, etc. Green also supports metaobjects. A metaobject is attached to an object to control the messages it receives. Every message sent to the object is redirected to a specific method of the metaobject. One can attach a metaobject to a *catch object* and then intercept the calling of a `throw` method, maybe changing the exception handling. A complete discussion of this topic is made by Guimarães [28].

The Green exception handling system (EHS) easily interacts with other language features because it is object-oriented. Then, subclassing is used to create specializations of *catch classes*, metaobjects can control exception handling (when attached to *catch objects*), the type system checks the correctness of the use of exceptions, `try-end` commands enhance the type of variable `exception`, polymorphism applies to catch objects, and so on. The close relationship EHS/language can be used to create further interactions between the EHS and other Green

[4] Metaobjects are cited at the conclusion of the article. They have been implemented in an older compiler.

[5] We use "idiom" to mean a Design Pattern specific to a language.

features. As an example, a factory class[6] [27] could supply catch objects for a program. It would be easy to change all error handling at once: one just needs to use another factory class object. For example, the program could have several options for error handling: a) issue the messages to the standard output/show the messages in a window; b) terminate/do not terminate the program.

These four options for error handling could be chosen by selecting a factory class object. In the beginning of codification, the chosen option could be standard output/terminate, which could evolve to window/do not terminate.

The Green EHS may look complex because:

- it uses a stack of catch objects in which a search for a handler is made;
- it uses an extra `exception` parameter used to throw exceptions. The compiler checks if an exception can be throw by examining the type of parameter `exception`. Each `try(catch)` statement enhances the type of `exception` by the type of catch.

However, these same elements are present, in a disguised form, in the exception system of Java/C++:

- the stack of catch objects corresponds to the dynamic nesting of `try` blocks at runtime in Java/C++. The search for a handler is made in the catch clauses of this dynamic chain of blocks;
- in Java, the compiler keeps a set of exceptions that may be thrown at a given point. At the first statement of a method, this set contains all classes following keyword `throws` that appears after the header of the method. This set increases at the beginning of a `try` block and decreases at the end. The growing of the set have a close relationship to the enhancement of the type of `exception` in Green.

Green just make the above concepts explicit by casting them into an object-oriented form. In particular, the search algorithms used by the Green EHS are essentially the same algorithms used in Java/C++.

Although Green uses object-oriented concepts in its EHS, there is still a gap between the Green EHS and object-oriented programming. The EHS employs a dynamic model in which *catch objects* are stacked at runtime, the type of `exception` changes dynamically, and searches for a `throw` method is made in a dynamic *stack of catch objects*.

Object-oriented programming models static hierarchies and does not fit quite well in representing dynamic behavior. That is the reason the search for a `throw` method takes into consideration the textual order in which `throw` methods are declared in a catch class. That is also the reason parameter `exception` is a restricted kind of variable — it cannot be assigned to another variable, for example. And its real type at runtime is the union of types of all active catch objects, which is a strange concept in object-oriented programming.

However, we do believe the mismatch between exception systems and object-oriented programming will be subject of future interesting research.

Acknowledgments. This work was partially financed by FAPESP under process number 99/13006-8.

References

[1] Dony, C. (1988) An Object-Oriented Exception Handling System for an Object-Oriented Language. *Proceedings of ECOOP 88*, Oslo, Norway, 15-17 August, pp. 146-161. Springer-Verlag, Berlin.

[2] Niemeyer, P. and Peck, J. (1997) *Exploring Java*. O'Reilly & Associates, Sebastopol.

[3] Stroustrup, B. (1991) *The C++ Programming Language*. Addison-Wesley, Reading MA.

[4] Guimarães, J. (1998) Reflection for Statically Typed Languages. *Proceedings of ECOOP 98*, Bruxelas, Belgium, 20-24 July, pp. 440-461, Springer-Verlag, Berlin.

[5] Guimarães, José de Oliveira. (2003) *The Green Language*. Available at http://www.dc.ufscar.br/~jose/green/green.htm.

[6] Harris, W. (1991) Contravariance for the Rest of Us. *Journal of Object-Oriented Programming*, 4(7), pp. 10-18.

[7] Garcia, A; Rubira, C; Romanovsky, A and Xu, J. (2001) A Comparative Study of Exception

[6] For short, a factory class creates and returns objects of other classes.

Handling Mechanisms for Building Dependable Object-Oriented Software. *Journal of System and Software*, 59(2), pp. 197-222.

[8] Miller, R. and Tripathi, A. (1997) Issues with Exception Handling in Object-Oriented Systems. *Proceedings of ECOOP 97*, Jyväskyl, Finnland, 9-13 June, pp. 85-103. Springer-Verlag, Berlin.

[9] Knudsen, J. L. (2003) *The BETA Language*. `http://www.mjolner.com/BETA`

[10] Knudsen, J. L. (2001) Fault Tolerance and Exception Handling in Beta. *Advances in Exception Handling Techniques*. pp. 1-17, Springer-Verlag, Berlin.

[11] Meyer, B. (1992) *Eiffel: The Language*. Prentice Hall, New York.

[12] Hof, M.; Möessenböck, H. and Pirkelbauer, P. (1997) Zero-Overhead Exception Handling Using Metaprogramming. *Proceedings of SOFSEM'97*, Milovy, Czech Republic, 22-29 November, pp. 423-431. Springer-Verlag, Berlin.

[13] Dony, C. (2001) A Fully Object-Oriented Exception Handling System: Rationale and Smalltalk Implementation. *Advances in Exception Handling Techniques*. LNCS Vol. 2022, pp. 18-38, Springer-Verlag, Berlin.

[14] Goldberg, A.; Robson, D. (1983) *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA.

[15] Mitchell, S.E.; Burns, A. and Wellings, A.J. (2001) MOPping up Exceptions. *Ada Letters*, XXI(3), 80-92.

[16] Berry, D.M. and Yemini, S. (1985) A Modular Verifiable Exception-Handling Mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), 214-243.

[17] Cui, Q. and Gannon, J. (1992) Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering*, 18(5), 393-401.

[18] Dony, C. (1990) Exception Handling and Object-Oriented Programming: Towards a Synthesis. *Proceedings of ECOOP/OOPSLA 90*, Ottawa, Canada, 21-25 October, pp. 322-330. ACM, New York.

[19] Dony, C. (1990) Improving Exception Handling with Object-Oriented Programming. *Proceeding of the 14th IEEE Computer Software and Application Conference*, Chicago, 31 October-2 November, pp. 36-42. IEEE, Los Alamitos,CA.

[20] Goodenough, J. B. (1975) Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12), pp. 683-696.

[21] Mikhailava, A. and Romanovsky, A. (2001) Supporting Evolution of Interface Exceptions. *Advances in Exception Handling Techniques*. LNCS Vol. 2022, pp. 94-110, Springer-Verlag, Berlin.

[22] Lippert, M. and Lopes, C. V. (2000) A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *Proceedings of the 22nd International Conference on Software Engineering*, Limmerick, Ireland, 4-11 June, pp. 418-427. ACM Press, New York.

[23] Kiczales, G; Lamping, J; Mendhekar, A; Maeda, C; Lopes, C; Loingtier, J. M. and Irwin, J. Aspect-Oriented Programming. *Proceedings of ECOOP 97*, Jyväskyl, Finnland, 9-13 June, pp. 220-242. Springer-Verlag, Berlin.

[24] Guimarães, J. O. and Johnson, R. Class Extension. (1996) *Proceedings of the Primeiro Simpósio Brasileiro de Linguagens de Programação*, Belo Horizonte, Brazil, 4-6 September, pp. 347-356. SBC, Belo Horizonte.

[25] Ossher, H. and Harrison, W. (1992) Combination of Inheritance Hierarchies. *Proceedings of OOPSLA 92*, Vancouver, Canada, 18-22 October, pp. 25-40. ACM Press, New York.

[26] Guimarães, J. O. (2001) An Idiom for Exception Treatment in C++ and Java. *Proceedings of the V Simpósio Brasileiro de Linguagens de Programação*, Curitiba, 23-25 May, pp. 268-275. SBC, Curitiba.

[27] Gamma, E; Helm, R; Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[28] Guimarães, José de Oliveira. (2003) *Exceptions and Meta-Level Programming in the Green Language*. Available at http://www.dc.ufscar.br/~jose/green/green.htm.