Error Recovery Using Threads

Mayer Goldberg Avi Shefi
{gmayer,shefi}@cs.bgu.ac.il
Department of Computer Science
Ben-Gurion University*

December 9, 2007

Abstract

In this work, we propose a practical approach to error-recovery in object-oriented programming languages. Error-recovery can be implemented using exceptions, but at the cost of having the programmer save and restore the state of the program manually, and restructuring the code so that the point of resumption is in tail position. This is tedious, prone to errors, and is non-compositional. In this work, we present an error-recovery package based on threads. Our approach prevents the programmer from having to manage state and restructure the code. The resulting programming style is compositional, and encourages incremental development of error-recovery code.

1 Error Handling & Recovery

Error-recovery handles exceptional situations and conditions that occur during execution of a program. Exceptional conditions occur as a result of an invalid state of a program, unexpected user input or behaviour, and genenraly any program state that might be encountered and is not defined as part of a programs functionality or purpose.

Error-recovery generalizes the domain of an operation. In this sense, error-recovery extends an operation's domain by making it usable in a wider variety of contexts. Goodenough [4] describes error-recovery as means for conveniently interleaving actions belonging to different levels of abstraction. Most modern programming languages support some form of exception handling, which provides a facitlity for non-local exit from a run-time context.

After performing error-recovery a program should resume normal execution. Garcia *et al* [3] describe two models for resuming control flow: (i) the *resumption* model, and (ii) the *termination* model. All of today's error-recovery mechanisms have adopted the termination model. Some additionaly provide the resumption model. Lyskov and Snyder [5] claim that the termination model is the most adequate model for resuming control flow due to its clearer semantics.

We suggest an approach where the code that controls the resumption of control-flow is placed within the error-recovery code, detached and apart from the place where the error is detected. The particular way of writing this code is what lets us capture error-recovery in a convenient and concise way.

In order to provide error-recovery using non-local jumps, we need to address two issues:

- Some state needs to be saved before the non-local exit, and restored before entering the resumption point. In object-oriented programming languages, this state can be stored in the throwable object. In functional programming languages this state corresponds to the environment of a closure that represents a continuation. Other implementations are possible as well.

*Department of Computer Science, Ben-Gurion University, P.O. Box 653, Beer Sheva 84105, ISRAEL.

- The resumption point, i.e., the point where control returns after recovery, can be reached from two places in the code, following the normal and exceptional paths of execution. Assuming a single thread of execution, this means that the context of the resumption point must be the same, regardless of the paths of execution. This is generally implemented by moving the resumption point to tail position, and reaching it from the normal and exceptional paths of execution by means of a tail call.

While it is possible to implement error-recovery using exception handling (or continuations), this requires the programmer to restructure the code and manage state explicitly. This is clumsy, and makes incremental development more difficult.

We propose a different approach to error-recovery, in which the thread that encounters an error is suspended and error-recovery code is executed in a separate thread. When error-recovery completes, the original thread is resumed and execution continues naturally. Our approach does not increase the total number of active threads. The cost of spawning such administrative, error-recovery threads can be amortized by using a thread pool.

Thread suspension and resumption provides an alternative approach to the two issues we raised above:

- The resumption point need not be in tail position, because awaking a thread resumes the natural execution path as specified on the control stack.

- The state of the thread is preserved during suspension, and is available after resumption, so the programmer need not be burdened with saving and restoring it manually.

We name this approach for error-recovery as *Threaded Exceptions*. The details of our design and implementation follow.

2 Implementation

We implemented a Java package that provides for threaded exceptions. Our package seperates the code that controls error-recovery and control-flow resumption from normal control-flow code. This seperation results from representing error-recovery in an object-oriented fashion. Exception handlers are dynamically defined during runtime using method calls to a service class. A handler is associated to an object through a hashtable, and is identified by the name of the error it is bound to.

Exceptions in Java are coded using a special *try-catch* clause. We associate a threaded exception handler to an exception name by calling `TE.addTEHandler`. This method takes three arguments: an object that may raise the exception, the name of the exception that the handler is bound to, and an exception handler. The handler is a class that inherits from `TEHandler` and implements its `call()` method. To raise a threaded exception we use `TE.throwTE(String)`, passing the name of the exception to be raised as an argument. Figure 1 shows a class diagram of the service class and `TEHandler`.

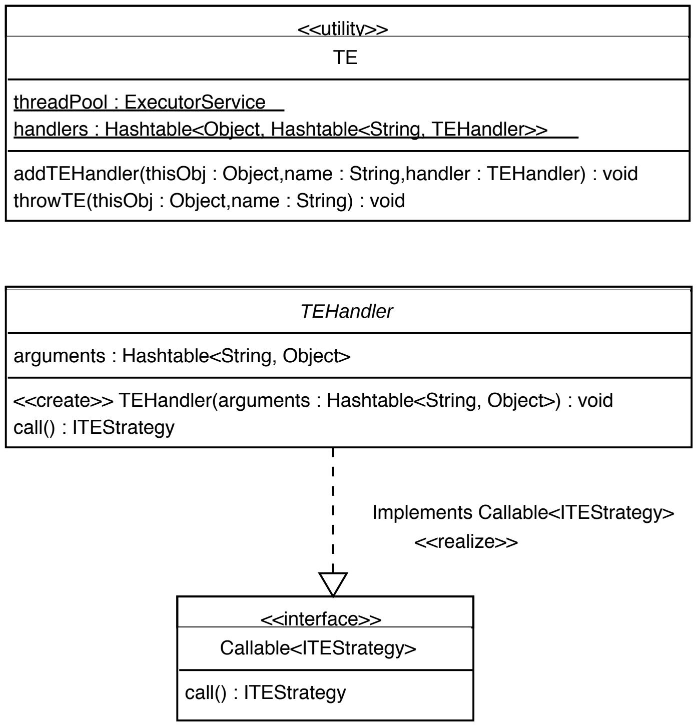
The `call()` method defined in the interface `TEHandler` is invoked in order to perform error-recovery. After error-recovery is finished, the program resumes normal control-flow. The method returns an implementation of the interface `ITEStrategy`, which defines the strategy by which control-flow is resumed.

We have implemented a pre-defined set of control-flow strategies to be easily used by the user:

- `TETerminateStrategy`, which represents the *termination* model.

- `TEContinueStrategy`, which represents the *resumption* model.

- `TEReThrowStrategy`, which invokes the exception handler again.



<<utility>>
TE
threadPool : ExecutorService
handlers : Hashtable<Object, Hashtable<String, TEHandler>>
addTEHandler(thisObj : Object,name : String,handler : TEHandler) : void
throwTE(thisObj : Object,name : String) : void
TEHandler
arguments : Hashtable<String, Object>
<<create>> TEHandler(arguments : Hashtable<String, Object>) : void
call() : ITEStrategy
Implements Callable<ITEStrategy>
<<realize>>
<<interface>>
Callable<ITEStrategy>
call() : ITEStrategy

Figure 1: Service class and TEHandler class diagram

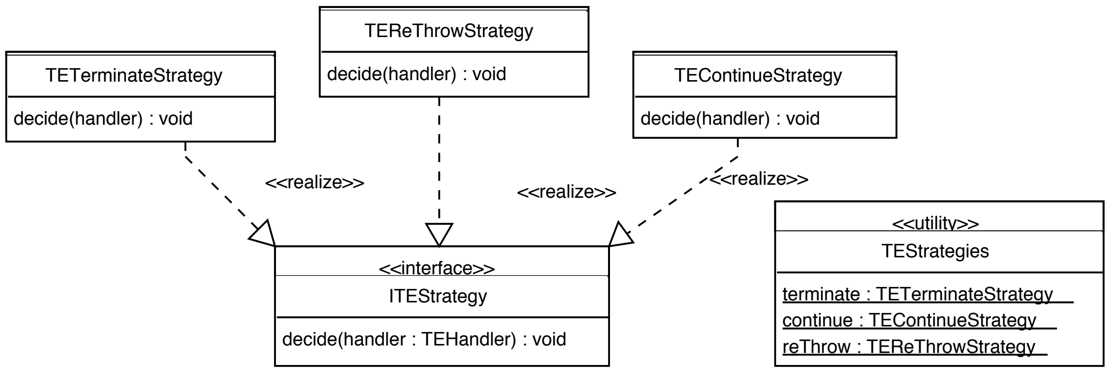


Figure 2: ITEStrategy class diagram

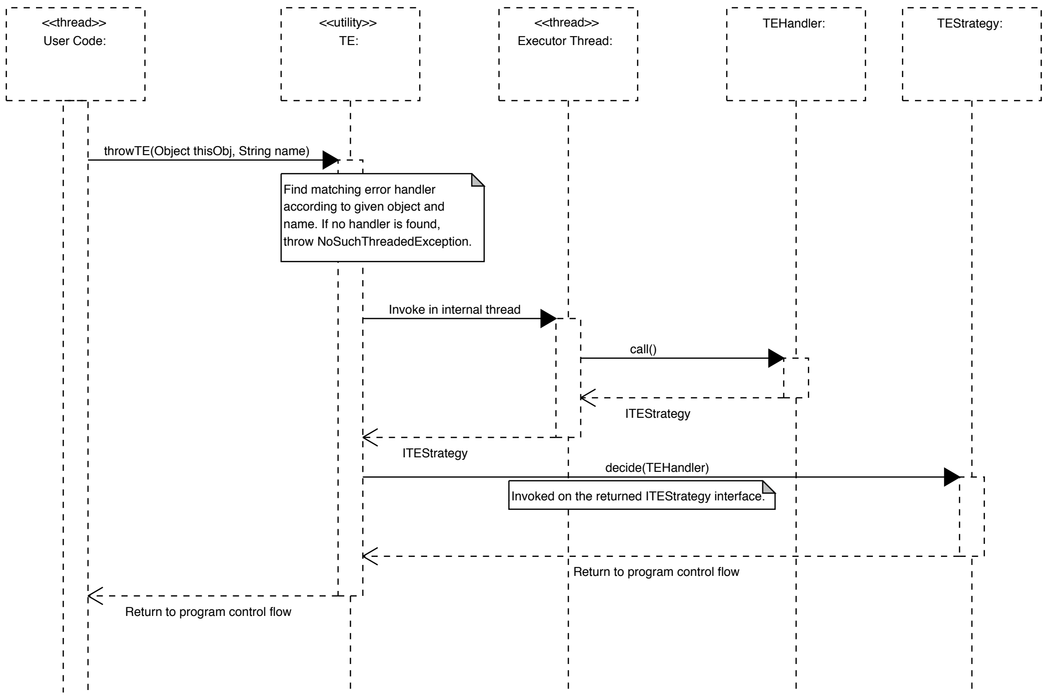

<<thread>>
User Code:
<<utility>>
TE:
<<thread>>
Executor Thread:
TEHandler:
TEStrategy:
throwTE(Object thisObj, String name)
Find matching error handler according to given object and name. If no handler is found, throw NoSuchThreadedException.
Invoke in internal thread
call()
ITEStrategy
ITEStrategy
decide(TEHandler)
Invoked on the returned ITEStrategy interface.
Return to program control flow
Return to program control flow

Figure 3: Throwing A Threaded Exception

`TETerminateStrategy` is implemented by raising a Java exception. Figure 2 shows a class diagram of `ITEStrategy` and the pre-defined set of control-flow models.

When a threaded exception is raised, the `call()` method of its associated handler is invoked. The method returns an implementation of the `ITEStrategy` interface, which defines a single method: `decide(TEHandler)`. This method represents the strategy by which control-flow resumes after error-recovery is performed. It takes a single argument, the threaded exception handler, to allow a possible re-invocation of the handler when needed. Figure 3 shows the sequence diagram of throwing a threaded exception and returning to normal control-flow.

3 Usage

We demonstrate our approach by simulating a read operation through a network connection. We require the read operation to be robust, and withstand repeated network failures, for example due to missed packets, etc. If the network connection cannot be recovered, the program must be terminated. We show how to handle this scenario using threaded exceptions. Using our approach, restoring the connection after network failures is decoupled from the read operation. This seperates exceptional code from normal control-flow code, and allows incremental development of error-recovery.

The simulated network connection is represented by the class `MyConnection`. The connection is assumed to have been established after object initialization. The method `readChunk()` reads data from the connection and returns it as a String. When disconnected, the object signals an abnormal state by raising an exception. A threaded exception is raised by calling `TE.throwTE(String)`, specifying the name of the error. Listing 1 demonstrates raising a threaded exception inside the `readChunk()` method, after detecting a connection failure.

Listing 1: Raising a threaded exception

```
public class MyConnection {
  //...
  public String readChunk() throws Exception {
    if (!this.isConnected())
      TE.throwTE(this, "ReadChunkException");
    return this.networkStream.read(buffer);
  }
}
```

Arguments are passed to the threaded exception handler using a hashtable. The handler attempts to re-establish the network connection and resume normal execution. The error-handler is implemented in *ReadChunkTEHandler*, a class derived from *TEHandler*. Listing 2 shows how control-flow is resumed from inside the error-handler using one of the pre-defined resumption strategies found in *TEStrategies*. The implementation details of the handler are irrelevant for this demonstration.

Listing 2: Resuming control flow

```
public class ReadChunkTEHandler extends TEHandler {
  public ITEStrategy call() {
    //...
    return TEStrategies.Continue;
  }
}
```

Our approach obviates the need to manage state manualy and allows programming in a compositional manner. The code is structured, and error-recovery is detached from the place where it occurs.

Listing 3 demonstrates binding a threaded exception handler to the type of error that `MyConnection` raises, followed by a sequence of read operations from the simulated network connection. If the network connection cannot be re-established, control-flow is terminated by the threaded-exception handler. When using the *termination* model, our package throws a normal Java exception. For this reason the code that may raise threaded exceptions is surrounded by a *try-catch* clause.

Listing 3: Binding a handler

```
MyConnection conn = new MyConnection();
Hashtable<String, Object> arguments = new Hashtable<String, Object();
arguments.put("conn", conn);
TE.addTEHandler(conn, "ReadChunkException",
  new ReadChunkTEHandler(arguments));

String chunk;
try {
  for (int i=0; i<10; i++) {
    chunk = conn.readChunk();
    System.out.println("Network␣read:␣" + chunk);
  }
} catch (Exception e) {...}
```

The resulting code is decoupled from normal control flow. The code that controls the resumption of control-flow is placed within error recovery. This clearly seperates between normal and exceptional code, and provides means for programming how normal control-flow is resumed.

3.1 Thread Pools

Spawning a thread per invocation of an exception-handler is costly in terms of both space and time. During normal program execution, multiple exceptions may be handled simultaneously. Moreover, a common usage of exceptions is to throw cascading exceptions from exception handlers, causing more overhead from a threading point of view. Thus, creating new threads for every exception is very expensive.

We use a thread pool to amortize the costs of spawning threads. The thread pool is initialized once during program startup. When an exception is thrown, its handler is queued for execution in the work queue of the thread pool. The exception is handled by the first available thread in the thread pool. In a highly multi-threaded application where there are multiple exceptions that are handled simultaneously, the thread pool expands accordingly and allocates more threads. This reduces the costs of handling exceptions, and results in a thread context-switch for every exception handler.

4 Related Work & Discussion

Research on various methods for error recovery has been conducted since the 1970's. Goodenough [4] analyzed various aspects and requirements for error recovery, and proposed a uniform notation for exception handling. Melliar-Smith and Randel [8] discussed the role of programmed exception handling in fault-tolerant systems, and offered a combined approach with recovery block structures. Zilles *et al* [12] presented a hardware-software solution, introducing a special hardware execution stack that is used specifically for exception handling.

Bagge *et al* [2] presented the *alert* concept, a uniform interface to all failure-mechanisms which seperates handling of exceptional situations from reporting it. Moessenboeck [6] presented a notation-based approach where an exception handler is defined using metaprogramming. Delegation of a raised error is carried by reflection using the type of the exception as a search criterion. Analyzing exception-handling mechanisms has been formalized by Garcia [3] who presented criterions for comparing different exception-handling mechanisms.

The first language to support some kind of exception handling is COBOL, dating to the late 1950's. Its exception-handling mechanism is not general-purpose, and handles a specific set of conditions such as end-of-file, overflow, underflow, etc. MacLISP was first to introduce catch-throw as an exception handling mechanism for handling exceptional conditions [9]. PL/I [7] and CLU [5] provide a single-level exception handling mechanism where an exception raised by a procedure has to be handled by its immediate caller.

Ada [11] treats exceptions as symbols and allows multi-level exception handling. It also introduces a special handler that catches all exceptions for which no specific handlers are provided. C++ [10] adopts the try-catch approach, and allows multi-level exception handling. Exception in C++ are data objects. Java [1] exceptions are objects that inherit from the class *Throwable*, and allows seperation between user exceptions and system exceptions (*checked/unchecked* exceptions). It also provides clean-up actions using the *finaly* clause.

References

[1] Arnold, K., G. J. H. D. *The Java Programming Language.* Addison-WEsley Longman Publishing Co. Inc., Boston, MA, USA, 2000.

[2] Bagge, A. H., David, V., Haveraaen, M., and Kalleberg, K. T. Stayin' alert: moulding failure and exceptions to your needs. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (New York, NY, USA, 2006), ACM Press, pp. 265–274.

[3] Garcia, A., Rubira, C., Romanovsky, A., and Xu, J. A comparative study of exception handling mechanisms for building dependable object oriented software: Journal of systems and software, 2001.

[4] Goodenough, J. B. Exception handling: issues and a proposed notation. *Commun. ACM 18*, 12 (1975), 683–696.

[5] Liskov, B., and Snyder, A. Exception handling in CLU. *IEEE Trans. Software Eng. 5*, 6 (1979), 546–558.

[6] M., M. H. H. Zero-overhead exception handling using metaprogramming. Tech. rep., Johannes Kepler Universitat Linz, 1997.

[7] MacLaren, M. D. Exception handling in PL/I. In *Proceedings of an ACM conference on Language design for reliable software* (1977), pp. 101–104.

[8] Melliar-Smith, P. M., and Randell, B. Software Reliability: The role of programmed exception handling. In *Proceedings of an ACM conference on Language design for reliable software* (1977), pp. 95–100.

[9] Moon, D. A. *The MacLisp Reference Manual.* MIT Project MAC, April 1974.

[10] Stroustrup, B., K.-A. Exception handling for C++. *The evolution of C++: language design in the marketplace of ideas* (1993), 137–171.

[11] Taft, S. T., and Duff, R. A., Eds. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, vol. 1246 of *Lecture Notes in Computer Science.* Springer, 1997.

[12] Zilles, C. B., Emer, J. S., and Sohi, G. S. The use of multithreading for exception handling. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 219–229.