Separating Exceptional Concerns

Anya Helene Bagge
Bergen Language Design Laboratory,
Dept. of Informatics, University of Bergen,
PB 7803, N-5020 Bergen, Norway
http://www.ii.uib.no/~anya/

***Abstract*—Traditional error handling mechanisms, including exceptions, have several weaknesses that interfere with maintainability, flexibility and genericity in software: Error code is tangled with normal code; reporting is tangled with handling; and generic code is locked into specific ways of reporting and handling errors. We need to deal with errors in a declarative way, where the concerns of errors, error reporting and error handling are separated and dealt with individually by the programmer.**

***Keywords*-Alerts; Exceptions; Guarding; Partiality; Separation of Concerns**

I. An Exceptional Tangle

A variety of techniques exist for reporting and dealing with errors,[1] such as special return values and exceptions. Even when exceptions are available, they are not always used: Alternative mechanisms are sometimes preferred for reasons of convenience, tradition, legacy or (real or imagined) resource constraints.

There are two main activities involved in handling errors: *reporting* that an error has occurred (from the callee side) and *handling* the error (on the caller's side). Common error reporting mechanisms include return values, error flags, and exceptions. Common error handling mechanisms include doing nothing, handling errors with conditional statements or jumps to error handling code, exception handlers, using a replacement value, or just aborting the program.

The variety in approaches to error handling causes several problems. First, the error handling mechanism must match the error reporting mechanism (you can't catch an exception by checking a return value). The reporting mechanism that works best for a library implementer may not be the best choice for the client; and for the client, different usage scenarios may work best with different error handling approaches.

Second, it may be difficult to apply modern techniques to legacy libraries; leaving library users with less convenient, inconsistent or unsafe error handling mechanisms.

Third, for some reporting mechanisms (such as return values), the default way of handling an error is to do nothing. This can make code prone to break in error situations; or cause the code to be littered with error-checking code.

[1]We'll use the word 'error' for any exceptional concern, since 'exception' has a very specific meaning in this context.

Finally, different implementations of an interface are locked into using the mechanism specified in the interface, even though they may have different error handling needs.[2]

In short, the error reporting mechanism is *tangled* with the error handling mechanism.

Another problem is tangling of error code with normal code. In Java, for example, **`try`**/**`catch`**-blocks are often scattered throughout the program, with similar **`catch`** clauses repeated in many methods. In C code, the situation can be worse, with much of the code dedicated to testing for errors. Aspects [1], [2] can solve this latter problem to some degree, untangling error code and normal code, but may not be able to fully untangle reporting and handling.

II. A Possible Solution

Alerts [3] provide separation of several error concerns: The callee can specify the partiality of functions using *guards* or preconditions, and the reporting of errors using *alert declarations*; and the caller can specify *handlers*, regardless of how errors are detected and reported. Handlers may be specified at various granularities, from expression to module level, thus making it easier to separate error code from normal code.

The key feature of alerts lies in the declaration of the reporting and handling mechanisms. Once these are explicitly visible in the code, rather than in the documentation, it becomes possible to automatically handle other reporting mechanisms than just normal exceptions, and to match different styles of reporting and handling.

In this section, alerts is shown more or less as described in the original paper [3], which was developed as a proof-of-concept of what can be done when errors are dealt with declaratively.

A. Declaring Alerts

In implementation code, we declare the error reporting mechanism in the signature of the operations, for instance, as conditions on the parameters or return values:

```
// null return means not found
Element get(Map map, Key key)
  alert NotFound if value == null;
```

[2]Particularly in concept programming, as we shall see later.

```
// divisor cannot be zero
int div(int a, int b) guard b != 0;
```

This states that a `NotFound` error has occurred if the result of `get` was `null`. We've used the return value mechanism to report errors; different implementations could declare **`throws`** `NotFound` instead, or use a **`guard`** to push detection to the caller.

In the case of `div`, we say that the divisor cannot be zero; or, rather, that the `div` function is only defined for non-zero divisors. Alternatively, we might throw a `DivisionByZero` exception in this case, or return `NaN`.

For legacy code, we need to be able to declare the error reporting separately. For example, in C, we might have a separate header file declaring the error behaviour of all the standard C library functions.

B. Handling Alerts

The user of the `get` function is concerned with how to handle `NotFound` errors, regardless of how they are reported. We might report an error and exit:

```
on NotFound in get {
  print("not found");
  exit(1);
};
```

This would install a handler in the current scoping level (local, function, class, module). Alternatively, we could substitute a suitable value at the expression level:

```
// use Jane Doe as default if get fails
name = get(persons, id) <:: "Jane Doe";
```

The alert system itself takes care of the glue between the reporting code and the handling code, and can automatically insert precondition and return code checks where necessary.

For division, we might set a sensible default which would abort the program. In other cases (for continuous functions) we might find a sensible approximation of division by zero using differentiation.

III. The Magnolia Language

We implemented a prototype of the system sketched above for C [3], but the infrastructure turned out to be difficult to use for larger scale experiments, so we have little practical experience with it. We are now revisiting the idea in the context of the experimental Magnolia language.

Magnolia allows for a close link between specification and code, using concepts.[3] A *concept* is an interface specification, describing types and operations and specifying their behaviour. Magnolia is built around the idea of concept-oriented programming, where everything is specified as concepts first, for which various interchangeable implementations may be provided. Concepts describing small pieces of functionality may be composed into larger concepts. This calls for flexibility in dealing with errors at both the specification level and in the program code.

IV. A Guarded Approach

In formal specification (or, when writing a Magnolia concept), specifying the particulars of error behaviour can drastically complicate reasoning, and may prematurely lock the design into a particular flavour of error handling. The use of *guards* is useful at this stage, either as preconditions in a design-by-contract style interface [5], or in the form of guarded algebras [6] in an algebraic specification.

A guarded operation is guaranteed to succeed as long as the guard holds. For example, we may state that the `get` operation will succeed as long as the requested key exists:

```
Element get(Map map, Key key)
  guard contains(map, key);
```

As long as we never use `get` with a non-existent key, we can ignore the entire possibility of errors, simplifying reasoning.

In practical programming, we can keep the guarded approach, forcing the programmer to always check `contains` first (leaving the result undefined otherwise), but this is not very useful in general. Typically, checking whether a map contains a key will involve actually looking up the key, making it rather wasteful to do it twice. Furthermore, we may not be able to determine in advance whether the operation will succeed without actually performing it; for instance, if `get` were to access a network database over an unreliable connection.[4]

Rather than programming directly with guards, we would like to use a traditional error reporting mechanism in the implementation of `get`, such as a null return. Hence, we are left with two different views of `get`:

- In the library interface or specification, getting non-existent keys is forbidden.
- In the library implementation, `get` returns null for non-existent keys.

The client uses only the interface to program against, and may not know which particular implementation is used. Any attempt to handle errors will have to rely on help from the language in connecting the error reporting code in the library with the error handling code.

Alert declarations, together with a suitable set of handler abstractions provide this connection: Each guard is connected to one or more alert declarations on the library implementation side. The client code deals with errors in terms of guards and alert handlers; the compiler must rewrite

[3] A similar idea [4], aimed primarily at constraining template parameters, was proposed for C++ 2011, but rejected. Development of C++ concepts continue however, and the problems discussed here also apply to concepts in C++.

[4] In this case, we might specify an opaque guard function, stating that `get` is undefined for a non-existent key, and in some other unspecified cases. On the implementation side, this would be dealt with using an exception, for example.

this code to use the error reporting mechanism supported by the library. In the final object code, no trace of the concept remains; even the guards (and in some cases the error handling code) might be compiled away.

While the above discussion is somewhat particular to concept-oriented programming, the issue also occurs to some degree with object-orientation [7]. For example, when specifying the exceptions of an interface or superclass in Java, one has to either declare a wide range of exceptions, providing less detail to the users, and possibly forcing them to deal with lots of exceptions that may never occur; or declare few or no exceptions, restricting the use of exceptions in the implementation or subclass.

V. Discussion

A declarative approach can provide increased flexibility in dealing with errors, decoupling the reporting of errors from the handling of errors.

In the original paper on alerts [3], we saw this largely as a convenience for the programmer; and as a feature that could fairly easily be patched on to a language like C, which provides no built-in exception support.

With concept programming, a system like alerts is more of a necessity: Two implementations of the same concept may be vastly different, making it infeasible to specify the concrete error mechanism in the concept. Furthermore, the concept user may have no idea which implementation will be used in the end, so client code cannot be tied to a particular implementation. The only entity that sees both the reporting and handling side is the compiler.

However, this problem is not restricted to concept programming, it can also be seen in generic code in mainstream languages; for example, with C++ template code calling code belonging to template parameters.

Guarding seems particularly useful from a specification and interface point of view. In program code, we might apply some reasoning to guarded calls, and figure out that errors can't happen (or always will happen, catching the error at compile time), and compile away the error code. Guards or pre/post conditions from an interface can also be applied directly in the implementation code (e.g., using aspects [1]). Normally, however, traditional error reporting and handling mechanisms are likely more suitable than guards, and linking guarding in specifications to traditional mechanisms in program code becomes important. We may also need to differentiate between multiple guards on the same operation, corresponding to different error conditions in the code.

Though it may be that traditional exceptions [8] would be good enough as a single reporting and handling mechanism, as far we can tell with Magnolia, this is not the case, as we will need to interface with existing C libraries and other legacy code, and to generate code for platforms (such as CUDA) where exceptions are unavailable or hard to implement.

The original alerts proposal includes constructs for abstracting over alerts and handlers; as well as a powerful handler model, somewhat similar to PL/I's `ON` [9] and Lisp's condition system [10]. This may turn out to be overly complex in practice, and is something we need more experience with.

Acknowledgements

This research has been partially funded by the Research Council of Norway.

References

[1] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 418–427.

[2] F. C. Filho, A. Garcia, and C. M. F. Rubira, "Error handling as an aspect," in *Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development*, ser. BPAOSD '07. New York, NY, USA: ACM, 2007.

[3] A. H. Bagge, V. David, M. Haveraaen, and K. T. Kalleberg, "Stayin' alert: Moulding failure and exceptions to your needs," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*. Portland, Oregon: ACM Press, October 2006.

[4] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in C++," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 291–310.

[5] B. Meyer, "Applying "Design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[6] M. Haveraaen and E. G. Wagner, "Guarded algebras: Disguising partiality so you won't know whether it's there," in *Recent Trends In Algebraic Development Techniques*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2000, vol. 1827, pp. 3–11.

[7] R. Miller and A. Tripathi, "Issues with exception handling in object-oriented systems," in *ECOOP'97 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, M. Aksit and S. Matsuoka, Eds. Springer Berlin / Heidelberg, 1997, vol. 1241, pp. 85–103.

[8] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, pp. 683–696, 1975.

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

[10] K. M. Pitman, "Condition handling in the Lisp language family." in *Advances in Exception Handling Techniques*. Springer-Verlag, 2000, pp. 39–59.