Refactoring Idiomatic Exception Handling in C++: Throwing and Catching Exceptions with Aspects

Michael Mortensen
Hewlett-Packard, Colorado State Univeristy
mortense@cs.colostate.edu

Sudipto Ghosh
Computer Science Department, Colorado State University
ghosh@cs.colostate.edu

Abstract

Aspect-oriented programming can be used to modularize cross-cutting concerns to improve the maintainability of large systems. Exceptions cross-cut legacy applications and are often implemented in idioms which cannot be globally enforced. We describe an aspect-oriented approach for throwing exceptions in place of the "return code idiom", and discuss using aspects to handle those exceptions in a modular way. We also describe challenges we encountered in implementing some exception-handling control flow strategies.

1. Introduction

We are investigating the use of aspects to modularize scattered code for detecting and handling errors in system calls, such as `fopen`. Aspects can modularize cross-cutting concerns that cannot be easily modularized by traditional object-oriented or procedural approaches [8].

Older programming languages, such as C, do not explicitly support exceptions. Instead they typically rely on an idiomatic approach for signaling and handling exceptions. One common idiomatic approach is the "return code idiom" [1], in which a special return code signals an exception has occurred. Even though C++ supports exceptions, C++ code may rely on legacy code and libraries that are written in C and use the return code idiom.

Bruntink, van Deursen, and Tourwe [1] describe several potential faults associated with the return code idiom. First, if the system does not check the return code of that function, the exception is ignored with potentially unpredictable behavior beyond that point. Second, the error code may need to be propagated up the call stack so that a series of calling functions must correctly check and signal exceptions through return values. Third, contextual information may need to be passed from the location of the exception to the function that should handle it. Such information is often managed through global variables and log files and may not be consistently implemented throughout an application.

Lippert and Lopes [11] discuss using aspects to convert errors in library functions and contracts for data types to exceptions. Spinczyk, Lohmann, and Urban [15] use AspectC++ to detect error-related return codes from MS Windows API functions. Throwing an exception whenever a special return code is detected benefits a system because exceptions can no longer be accidentally ignored: failure to handle an exception causes program termination. Filho, Rubira and Garcia [4] implement exception handling with aspects for applications that already use exceptions.

Although exception throwing can be modularized as a single aspect, if try/catch blocks are added to the original code, then these blocks are still just as scattered as checking the return code was. Since the code in the scattered catch blocks may be implementing the same or similar error-handling code, we modularize error signaling and error-handling with aspects. The legacy application we are refactoring, however, implements five different strategies for handling return code errors. Thus, exception handling requires multiple disjoint aspects, each with slightly different requirements.

We examine the benefits of our approach over the idiomatic approach and also highlight two challenges. The first challenge is that since there are different ways that exceptions need to be handled in the code, we cannot modularize exception handling as a single aspect. Second, some of the aspects use pointcuts that enumerate all associated functions. Although such an approach does modularize the exception handling code for each exception handling strategy, such pointcuts are fragile and could lead to errors during evolution [10].

The rest of the paper is organized as follows. In Section 2 we describe the application-specific return code idiom we are refactoring, including a single aspect for converting return codes to exceptions. Section 3 describes the different exception-handling strategies used by the application and how aspects can implement them. Potential extensions to AspectC++ that could better address the challenges we encountered are described in Section 4. Related work is described in Section 5, followed by our conclusions in Section 6.

2. Replacing the Return Code Idiom

We are refactoring a CAD tool, the `PowerAnalyzer`, used for estimating circuit power. It consists of about 12,000 lines of C++ code. The `PowerAnalyzer` contains a function that is similar to the standard C `fopen` function. Like `fopen`, this function uses the return code idiom, signalling failure by returning a NULL pointer. The type of error is indicated through a global variable, `errno`, which can be accessed by modules that include the `errno.h` header file. Because our function behaves like `fopen`, we will describe the problem in terms of `fopen` in the remainder of this paper.

2.1 Converting return codes to exceptions

Detecting a failed `fopen` and converting it to an exception is simple in AspectC++. We define a struct for the object to be thrown and then have an aspect that uses after advice to check the return value of `fopen`, as shown below.

```
// The exception object
struct FileNotFound {
 char name[2048];
 char mode[5];
 int  code;

 FileNotFound(const char *filename,
    const char *file_mode)
 {
   strcpy(name, filename);
   strcpy(mode, file_mode);
 }
};

//Aspect to check fopen result
aspect Excepter {
 pointcut FOpen()=call("% fopen(...)");
 advice FOpen() : after() {
  //get the result of the fopen call
  void *ptr = *tjp->result();
  if( ptr==NULL) {
   const char *filename =
      *( (const char **) tjp->arg(0) );
   const char *mode =
      *( (const char **) tjp->arg(1) );
   throw FileNotFound(filename,mode);
  }
 }
};
```

The Excepter advice defines `after` advice which uses the AspectC++ function `tjp->result()` to inspect the return value of `fopen`. The aspect throws a `FileNotFound` exception anytime `fopen` returns NULL.

The exception throwing code is contained in a single advice body, and its pointcut is based on a single function name (`fopen`). The aspect provides a modular way of adding exceptions using a pointcut that is easy to specify and maintain since it is based only on the name of the function (`fopen`) that triggers the error.

3. Aspectualizing the Exception Handling Strategies

3.1 Scattered exception handling

The calls to the `fopen`-like function and accompanying error-handling code are scattered across 19 different call sites. The `PowerAnalyzer` opens many different types of files: design data files, user configuration files, output results files, optional debug files, and so forth.

Some of these files are essential for the program to run, while others are optional or only used in certain modes. Because of these different file types, the `PowerAnalyzer` implements 5 different strategies for handling return code errors.

The strategy used within each function that calls `fopen` (referred to as the *caller* or *calling function*) represents a design decision, since the strategy depends on the type of file being opened and the algorithmic step being performed by the program. Each strategy has one of three control flow semantics. These control flow semantics are exiting from the program, returning from the function that calls `fopen`, and continuing past the failed `fopen` call.

The five strategies implemented in the `PowerAnalyzer` are based on the control flow they implement. These strategies are:

1. Exit program due to fatal error.
2. Warn user; return from caller.
3. Check for user configuration; either warn and return from caller or exit program.
4. Return from caller.
5. Warn user but continue inside caller.

We describe each below, and discuss how an aspect can replace the existing implementation.

3.2 Exit program due to fatal error

Fatal errors are idiomatically handled by calling a function, `FatalError`, that prints a message and exits the program.

```
 fp = fopen(file, "r");
 if(!fp) {
   FatalError("Could not open", file);
 }
```

The code that calls `FatalError` when the return value (`fp`) is `NULL` can be aspectualized by adding a try/catch block in the highest level function of the program (`main`). In addition, an aspect can use around advice that encloses `main` in a try/catch block:

```
aspect CallCatcher {
 pointcut Main() =
    execution("% main(...)");
 advice Main() : around()
 {
  try {
   tjp->proceed();
  }
  catch (FileNotFound e) {
   std::cerr << "Error opening file: "
    << e.name << " in mode: " << e.mode
    << " from "<<JoinPoint::signature()
    << std::endl;
    exit(1);
  }
 }
};
```

Analysis. Using an aspect around `main` does not reduce code scattering since `main` occurs only once in any program. However, because we can add the advice in the `CallCatcher` aspect to the aspect that also throws exceptions, we can modularize throwing and handling exceptions as a single aspect.

The aspect-oriented approach for the fatal error strategy enables removing the error-handling code from each call to `fopen`. Both the advice for catching and throwing exceptions are each based on a single function name: `fopen` for throwing exceptions, and `main` for catching them. Thus, aspects provide a modular way of implementing this strategy using pointcuts that are easy to specify and maintain.

3.3 Warn user; return from caller

This strategy prints a warning to the user and returns from the caller. The functionality of the caller is skipped, but the program does not exit. Typically, the code is structured like this:

```
void caller()
{
 fp = fopen(config_file, "r");
 if(!fp) {
  Logger("Warning, could not open ",config_file);
  return;
 }
 //rest of caller...
}
```

The `Excepter` aspect from Section 2.1 will throw an exception if `fopen` returns a `NULL` pointer. In order to continue as if only a return had happened from inside the caller, the exception must be caught immediately outside the caller. If the exception is not caught or is caught in `main`, the program will terminate.

We can use an aspect to catch this exception and return from the calling function, but to do so requires specifying the name of the caller. If the calling functions have a common naming convention, then the pointcut can use a regular expression to match them. However, in the `PowerAnalyzer`, exception handling is scattered across many functions with different naming conventions. This requires that we enumerate all of them as a list. For example, if the callers are `a`, `b`, and `c`, then the `WarnAndReturn` aspect below implements the warn and return strategy:

```
aspect WarnAndReturn {
 pointcut throwing_funcs() =
   execution("% a(...)")
   || execution("% b(...)")
   || execution("% c(...)");
 advice throwing_funcs() : around()
 {
  try {
   tjp->proceed();
  }
  catch (myFileNotFound e) {
   std::cerr << "Error opening file: "
    << e.name << " in mode: " << e.mode
    << " from "<<JoinPoint::signature()
    << std::endl;
  }
 }
};
```

Analysis. The warn and return aspect modularizes the `fopen` error handling code to one location. If the strategy itself needed changes, the `WarnAndReturn` aspect's advice could be updated in one place, rather than modifying all the callers that implement this strategy.

Unfortunately, the pointcut, `throwing_funcs`, is fragile: all functions using the strategy must be enumerated using the pointcut. If a function name is changed or if a new function is added, we must remember to update the aspect. Koppen and Stoerzer [10] define fragile pointcuts as those that have high name-based coupling between aspects and core concerns and may be broken by non-local changes during evolution. In the `PowerAnalyzer`, the fragile pointcut introduces a risk that the exception may not be caught. This would result in an incorrect strategy being used: the exception would be caught by outermost block assocated with `main` as described in Section 3.2.

3.4 Check user configuration, warn and return or exit

The `PowerAnalyzer` allows users to specify that some errors should not result in program termination. This strategy checks for the user configuration; if the user has specified continuing in the presence of issues, then the caller emits a warning and returns (2nd strategy). Otherwise, the program exits (first strategy).

If the user did not specify to continue in spite of errors then the aspect for this strategy can print a message and call `exit` to terminate the program. If the user did specify continuing past these errors, then the control flow is the same as the strategy described in section 3.3.

Thus, the new aspect, `ReturnOrExit`, is very similar to that of Section 3.3 but adds configuration checking and a branch that calls `exit`. Whether or not the user specified continuing past errors is represented below as the variable `user_config_continue`. Here is the `ReturnOrExit` aspect:

```
aspect ReturnOrExit {
 pointcut throwing_funcs() =
   execution("% x(...)")
   || execution("% y(...)")
   || execution("% z(...)");
 advice throwing_funcs() : around()
 {
  try {
   tjp->proceed();
  }
  catch (myFileNotFound e) {
   if( user_config_continue ) {
    std::cerr << "(WARNING) Could not open file: "
     << e.name << " in mode: " << e.mode
     << " from "<<JoinPoint::signature()
     << std::endl;
   }
   else {
    std::cerr << "Error opening file: "
     << e.name << " in mode: " << e.mode
     << " from "<<JoinPoint::signature()
     << std::endl;
     exit(1);
   }
  }
 }
};
```

Analysis. This aspect is more complex than the `WarnAndReturn` aspect from Section 3.3 since its advice catches the exception and then either emits a warning message (outside the caller, being equivalent to a return) or calls exit. In terms of maintainability, it has the same problem: it requires a pointcut that is a list of functions (shown in the aspect above as `x`, `y`, and `z`) so that it can catch the exception just outside the function. Thus, while the aspect does modularize scattered code into a single advice body, the pointcut itself may be difficult to maintain.

3.5 No warning, error indicated through return value.

This strategy does not warn the user about failures and is used when a large set of files is being processed. Success or failure reading the file is only indicated by the return value, and no warning is emitted. The basic code structure is:

```
int caller(char *file)
{
 fp = fopen(file, "r");
 if(fp) {
   //process file...
   return 0; //no error
 }
 else
   return 1; //error opening file
 }
}
```

Like the `WarnAndReturn` aspect from Section 3.3, we need around advice just outside the caller. The key difference is that the advice uses `thisJoinPoint->result()` to get a pointer to the retun value (`result`), which is used to change the return value if the file cannot be opened. The `CatchAndReturn` aspect, like the `WarnAndReturn` aspect, catches the exception so that the program continues execution after the caller.

```
aspect CatchAndReturn {
 pointcut throwing_funcs() =
  execution("int read_circuits(...)")
  || execution("int process_config_files(...)");

 advice throwing_funcs() : around()
 {
  int *result = (int*) thisJoinPoint->result();
  try {
   tjp->proceed(); //just call what we caught...
  }
  catch (myFileNotFound e) {
   std::cerr << "(WARNING) Could not open file: "
    << e.name << " in mode: " << e.mode
    << " from "<<JoinPoint::signature()
    << std::endl;
    *result = 1; //return 1(error) using result ptr
  }
 }
};
```

Analysis. This strategy has the same weakness as the strategy of Section 3.3: the pointcut is a list of all calling functions. In addition, if the `PowerAnalyzer` had functions that used different return values to indicate errors in the caller of `fopen` (e.g. return 0 in one caller and return -1 in another), then separate pointcuts would be needed, each enumerating a list of function calls. Handling a group of functions that have different return types would also add complexity, although advice in AspectC++ can use C++ templates so that code is generated for each different type at compile time [13].

3.6 Warn user; continue inside caller

For this strategy, we want to emit a warning but continue inside the caller. One approach would be to use aspects to throw the exception, but to enclose calls to `fopen` in a try/catch block *inside* each calling function:

```
void caller()
{
 try {
  fp = fopen(config_file, "r");
 }
 catch ( FileNotFound e) {
  Logger("Warning, could not open ",
    e.name());
 }
 //Keep going after exception
}
```

However, the exception handling code is *still* scattered in all the callers. Lippert and Lopes [11] also describe this problem: "AspectJ 0.4 does not provide support for capturing the catching of exceptions inside method boundaries." Exceptions caught by around advice are caught after the intercepted method body since the pointcuts available in AspectJ and AspectC++ are method calls rather than code blocks within the methods.

However, we can continue inside the caller if we catch the exception around the call to `fopen` instead of around the caller, using the `cflow` mechanism of AspectC++ (which AspectJ also has) to select the functions that should implement this strategy. The advice would be the same as in the `WarnAndReturn` aspect, but the pointcut would be defined differently using `cflow` with the name of the caller. An example pointcut definition for the `InnerCallCatcher` aspect is shown below:

```
aspect InnerCallCatcher {
 pointcut Fopen() = call("% fopen(...)");
 pointcut local_catch() = Fopen()
  && cflow(execution("% d(void)"));
```

The `local_catch` pointcut of the `InnerCallCatcher` aspect can wrap calls to `fopen` that occur within the execution of the function `d`. By catching the exception around `fopen`, the rest of the function `d` continues execution after the catch block in the advice executes.

Analysis. Unfortunately, this pointcut approach again becomes difficult to maintain as more functions need to be specified. For example, to add an extra caller, `e`, again requires a list of pointcuts:

```
 pointcut Fopen() = call("\% fopen(...)");
 pointcut local_catch1() = Fopen()
  && cflow(execution("\% d()"));
 pointcut local_catch2() = Fopen()
  && cflow(execution("\% e()"));
 pointcut local_catches() = local_catch1()
  || local_catch2();
```

The `cflow` construct introduces some run-time overhead [12]. In addition, the `cflow` pointcut matches *any* call to `fopen` that happens below the specified caller (`d`). Using the `within` construct would limit the pointcut to calls that occur immediately within the named functions. This would improve performance and limit the scope to just `fopen` calls within the specified function, but still requires a fragile list of pointcuts. The `cflow` approach would help if `d` called several functions that called `fopen` since we could specify just the top-most level (`d`) instead of the intermediate functions. In Section 3.7 below, we describe a better solution for this strategy.

3.7 Using a facade to implement the "Warn user; continue inside caller" strategy

We can avoid the list of functions and use of `cflow` for the "Warn user; continue inside caller" strategy. Instead, we can create a facade function, `fopen_continue`, that wraps the call to `fopen`.

```
FILE* fopen_continue(
  const char* filename,
  const char* mode) {
    return fopen(filename, mode);
}
```

In functions that implement this strategy, we replace the call to `fopen` with a call to the facade, `fopen_continue`. We can now implement an aspect that uses `around` advice so that a try/catch block exists around the `fopen_continue` function. Thus, an exception thrown from a failed `fopen` call is caught and handled inside `fopen_continue`, allowing the function that calls `fopen_continue` to continue execution. This does require changing all calls to `fopen` in functions that implement the third strategy, but it makes the strategy explicit and avoids the fragile pointcut.

Unfortunately, using a facade for `fopen` does not help with the strategies described in Sections 3.2, 3.3, and 3.4 where we want to return from the function that calls `fopen`. That is because the catch block needs to happen outside the caller so that the upper scope (above the caller) continues. Functions that need to return after `fopen` fails need this try/catch structure:

```
upper_scope()
{
 try {
  caller()
  {
    fopen(...);
  }
  catch {
    //do something but don't exit, so
    //that we continue in upper_scope
  }
 //rest of upper_scope
}
```

Analysis. Using a facade around `fopen` helps us catch exceptions inside the caller, but not around the caller. Thus, it is a better solution than Section 3.6, but does not provide a better solution for the strategies of Section 3.2, Section 3.3, or Section 3.4. In our refactoring, we elected to use the facade/aspect approach for the third strategy because of the simpler pointcut.

If a facade-based approach could be used for all strategies, an additional benefit would be that each strategy would be explicitly declared by use of the facade, with the aspect providing the modular implementation of each strategy. Using facades in this way requires changing all calls to `fopen` to an `fopen` facade. Since we had to make changes at each callsite to remove obsolete error handling code when aspectualizing the `PowerAnalyzer`, we did not consider changing calls from `fopen` to use the facade to be a significant burden.

3.8 PowerAnalyzer Refactoring

We manually refactored the `PowerAnalyzer` to use aspects for throwing and catching exceptions. The results of our technique are shown below in Table 1. The exception strategy is shown in the first column, and the number of occurrences of each strategy are given in the second column (#Occ). The pointcut column contains 'Simple' if the strategy in the first column was implemented as a single pointcut or contains 'List' if the pointcut was a fragile, list-based pointcut.

Table 1. PowerAnalyzer exception strategies

Strategy	# Occ	Pointcut
1. Fatal error, exit	7	Simple
2. Warn user, return	2	List
3. Check config, return/exit	5	List
4. Indicate with return val	3	List
5. Warn user, continue	2	Simple

In total, there were 19 calls to `fopen`, with each having 3-5 lines of code after the call to implement handling `fopen` errors. Using aspects for `fopen` error-handling replaced about 80 lines of code with 6 aspects, for a total code reduction of 40 lines. Having to implement 5 different strategies with aspects limited the code reduction. Filho, Rubira and Garcia [4] also found that complex applications with multiple exception-handling strategies limited code reduction. Even with the code reduction, a drawback is that three of the five aspect-based strategies were implemented with a fragile pointcut.

In spite of the pointcut maintenance issue, one benefit of aspects is that the aspects throw exceptions that will result in program termination if the pointcuts are not correct. Although this is an abrupt result, it avoids accidentally ignoring `fopen`-related errors, which is a common fault with the return code idiom.

One way to validate the pointcuts in our exception-related advice would be to use statement coverage of regression tests to ensure that *all* `fopen`-related errors were tested. This is often difficult in practice, since testing typically focuses on primary application functionality rather than code related to exceptions and since many root causes of exceptions are difficult to generate [1].

4. Annotations and Pointcut Extensions

In this section we describe two extensions to AspectC++ that would enable an aspect-oriented solution that did not use fragile pointcuts. The first approach, annotation-based weaving, is currently available in AspectJ [9]. The second approach utilizes pointcut expressions based on more complex program flow analysis.

4.1 Annotations

Although not currently available in AspectC++, annotations and annotation-based weaving could be used to replace the pointcut lists. Annotations do not allow us to modularize the specification of all the join points in one location: each location in the code of a particular pattern will require an annotation. However, renaming methods to match an aspect's pointcut has the same limitation and may also cause name clashes if a method is to be advised by multiple aspects that all want to use name-based pointcuts. Even though the annotations are scattered in the code, annotations are preferable to a pointcut implemented as a list of functions. Using a facade in Section 3.7 is equivalent to annotation-based weaving since the facade name is being used as the weave target for a specific aspect strategy. Like the facade, annotations document the code intent. In addition, name changes to the advised function will not accidentally break the aspect's pointcut.

Were annotations to be supported, the method that calls `fopen` could be annotated to indicate the aspect that will provide the exception strategy. The aspect could then provide around advice for the caller of `fopen` by advising at the annotation. If `fopen` throws an exception, it would be caught immediately outside the annotated caller. The aspects in Section 3 that used fragile lists of pointcuts could be implemented with a single annotation-based pointcut.

4.2 Complex control flow pointcuts

More powerful control-flow pointcut specifications are an alternative to annotations. For example, we could specify that we want to have advice around all methods that call `fopen` without enumerating them in a fragile pointcut list. Cazzola, Pini, and Ancona [2] proposed higher level pointcuts, such as 'all methods that are setters' rather than relying on naming conventions for name-based pointcuts for setters, in order to have aspects that are more robust during evolution.

If we had a more expressive pointcut that specified all callers of `fopen` in this application, we could avoid pointcut lists. Since, however, there are multiple strategies for exceptions present in this application, we must still distinguish between when to exit due to a fatal error, when to warn and continue, and so forth. One approach would be to rename the `fopen` calls based on the exception strategy for that call (i.e. `fopen_or_exit`, `fopen_or_warn`) so that we could select all callers of `fopen_or_exit` with a single pointcut. While this is more modular than list-based pointcuts, the end result is similar to annotation-based weaving, since we must still make a change in the `fopen` callsite.

5. Related Work

5.1 Checked and unchecked exceptions

C++ and AspectC++ do not have checked exceptions like AspectJ. AspectJ and Java perform static checking of exceptions to ensure that method calls that can result in exceptions either catch those exceptions or declare that the methods may throw them. To allow an aspect to throw an exception not declared by the method, an aspect in AspectJ can specify that an exception "if thrown at a join point, should bypass Java's usual static exception checking system and instead be thrown as a `org.aspectj.lang.SoftException`, which is subtype of `RuntimeException` and thus does not need to be declared." [1].

[1] `http://www.eclipse.org/aspectj/doc/released/progguide/semantics-declare.html#softened-exceptions`

In C++, exceptions do not have to be declared by each function or method and are not checked by the compiler [16]. This allows aspects in AspectC++ to throw an exception that was not originally thrown by the function, and allows changing where exceptions are caught.

5.2 Contracts and Obliviousness

Some might argue that we have changed the contract of `fopen` since it now throws an exception in the aspectualized program. Anytime `fopen` is called, either an aspect or the calling code will need to catch an exception or the program will terminate. Java has three types of exception-handling blocks: try-catch, try-catch-finally, and try-finally. Filho, Rubira and Garcia [4] reported that handling all three increased the number of aspects required when aspectualizing exception handling.

A related issue to the notion of the contract of `fopen` is obliviousness. An aspect that throws exceptions when `fopen` fails to open a file is introducing a change that the `fopen` code is oblivious to, but which must be handled by the callers of `fopen` or else the exception will not be caught, resulting in program termination. While obliviousness has been discussed as a potential benefit of aspect-oriented programming [5], in this case developers need to be aware of the new exceptions being introduced in order to implement an exception handling strategy. Using exceptions improves error handling by avoiding accidental ignoring of failures, but at the cost of mandating exception handling and risking program termination if some cases are not caught.

Griswold et al. [6] report that developing aspects and core concerns that were oblivious to one another led to “programs that were unnecessarily hard to develop, understand, and change.” One reason they reached this conclusion was that changes to the code base that seemed harmless could change what join points matched. Our aspects that create pointcuts as lists of function names would similarly be affected by changes in the code base.

5.3 Aspect-oriented exception approaches

Lippert and Lopes [11] use aspects for exceptions, but focus on throwing exceptions, both for library functions, which are like our `fopen` example, and also for contracts. They note that handling exceptions can be difficult because exceptions thrown by aspects must happen at method boundaries, while existing try/catch blocks may exist within method boundaries.

Spinczyk, Lohmann and Urban [13, 15] focus on using template-based advice so that a single aspect can detect an exception signalled by the return code idiom across a group of methods with different return data types. Their code works by detecting such things as NULL pointers, false values from `bool` functions, and so forth. They do not focus on catching the exceptions, leaving that to the application code.

Filho, Rubira and Garcia [3, 4] implement exception-handling in AspectJ. They remove existing try/catch blocks from a system that already throws exceptions in the core Java code. They also found that existing legacy systems tend to have many non-uniform complex strategies, which makes moving them to modular aspects more difficult. Our work differs from theirs by using aspects for both throwing and catching exceptions. In addition, we are refactoring legacy code that did not use exceptions, but instead used the return code idiom.

6. Conclusions and Future Work

We have demonstrated how the return code idiom for exceptions can be refactored with aspects to use exceptions. Aspects can be used not only to replace return code with thrown exceptions, but also to manage how and where those exceptions are caught and dealt with. We found that the aspect-oriented approach reduces source code size and modularizes the scattered code for implementing each exception strategy into an aspect.

The major drawback to our aspect-oriented approach is that some strategies required the use of pointcuts that were lists of functions. While this approach does work, it may be viewed as fragile for long term maintenance since changing function names or creating new functions that should be in these lists can result in incorrect exception behavior [10].

In addition to AspectC++ language extensions described in Section 4, another approach would be to use naming conventions for the functions that were part of pointcut lists. Although this approach would work in isolation, we did not pursue it because we are already using a naming convention for a Timer aspect [14]. Since functions that are advised by the Timer aspect could also use `fopen`, this would result in functions needing a name that matches *both* pointcut naming patterns. Clearly, as more aspects are added to this application, such a name-based solution becomes difficult or impossible. Tourwé, Brichau and Gybels[17] describe this problem as part of the AOSD-evolution paradox.

References

[1] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE ’06: Proceeding of the 28th International Conference on Software Engineering*, pages 242–251, New York, NY, USA, 2006. ACM Press.

[2] W. Cazzola, S. Pini, and M. Ancona. Design-Based Pointcuts Robustness Against Software Evolution. In W. Cazzola, S. Chiba, Y. Coady, and G. Saake, editors, *Proceedings of the 3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE’06)*, in 20th European Conference on Object-Oriented Programming (ECOOP’06), pages 35–45, 2006.

[3] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhao, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT ’06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152–162, New York, NY, USA, 2006. ACM Press.

[4] F. C. Filho, C. M. F. Rubira, and A. Garcia. A quantitative study on the aspectization of exception handling. In *In ECOOP’2005 Workshop on Exception Handling in Object-Oriented Systems*, pages 137–149, July 2005.

[5] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[6] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[7] K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, editors. *European Interactive Workshop on Aspects in Software (EIWAS)*, Sept. 2004.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox PARC, Feb. 1997.

[9] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP*, pages 195–213, 2005.

[10] C. Koppen and M. Störzer. PCDiff: Attacking the fragile pointcut problem. In Gybels et al. [7].

[11] M. Lippert and C. V. Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427. ACM Press, 2000.

[12] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schrder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of EuroSys ’06*, April 2006.

[13] D. Lohmann and O. Spinczyk. On typesafe aspect implementations in C++. In Gybels et al. [7].

[14] M. Mortensen and S. Ghosh. Using aspects with object-oriented frameworks. In *AOSD '06: 5th International Conference on Aspect-oriented Software Development Industry Track*, pages 9–17, March 2006.

[15] O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP extension for C++. In *Software Developers Journal*, number 7, pages 68–74. Software-Sydawnicto Sp. z o.o.

[16] B. Stroustup. *The Design and Evolution of C++*. Addison Wesley, April 1994.

[17] T. Tourwé, J. Brichau, and K. Gybels. the existence of the AOSD-evolution paradox. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*, Mar 2003.