Computational Error Handling as Aspects: A Case Study

Zhanqi Cui, Linzhang Wang, Huigen Liu, Xuandong Li


Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.
Computational Error Handling as Aspects: A Case Study

Zhanqi Cui\(^1, 2\), Linzhong Wang\(^1, 2\), Huigen Liu\(^3\) and Xuandong Li\(^1, 2\)

\(^1\)State Key Laboratory of Novel Software Technology
\(^2\)Department of Computer Science and Technology, Nanjing University
\(^3\)Department of Astronomy, Nanjing University
Nanjing, Jiangsu, P. R. China 210093
zqcui@seg.nju.edu.cn; \{izwang,lxd\}@nju.edu.cn

ABSTRACT
Computational error handling is vitally important in scientific computing programs. Traditional approaches tangle the error handling concerns with the primary functional codes. This paper describes our empirical study of using AspectJ to refactor a real-world Satellite Orbit Forecasting (SOF) system, and assesses the experiences gained from applying aspect-oriented techniques. The result shows that the AspectJ version improves the modularity and maintainability of the program by encapsulating computational error handling policies as aspects without noticeable compromise in performance.

Categories and Subject Descriptors
D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms
Languages

1. INTRODUCTION
Computation has been considered as the third branch of scientific research, along with theory and experiment. Many theoretical math problems can only be solved by using the computation method, which is also known as numerical analysis. Numerical analysis focuses on figuring out mathematical problems in scientific and technical researches by the means of computing with computers. The rapid growth of computing power and sophisticated computation methodologies have permitted the utilization of computation methods in application fields including bioinformatics, computational chemistry, new drug discovery, novel materials design, and aerospace technology. However, the introduction of errors is unavoidable in scientific computing. Thus, computational error handling is one of the most important things concerned by both domain experts and programmers.

Problem Statement. The design and implementation of computational error handling policies are critical concerns in scientific computing programs, because computational errors directly affect the accuracy and reliability of the systems, especially for safety-critical applications. Computational errors could be introduced at every computing statement in a scientific program. Traditionally, the implementation of corresponding error handling policies are applied at every error-produced position in the program. They cut across the primary functionalities of the program. As a result, the traditional solution of error handling causes the tangling problem \(1\), i.e., error handling code is scattered and repeated in many places, which make the program difficult to reason about and maintain. Moreover, error handling policies need to be optimized and adopted separately with the primary functional features to meet changing requirements of precision and performance. This will make the tangling and maintenance problem even worse. If error handling policies were separated from the tangled code and localized into encapsulated modules, the program should be much easier to be understood and maintained.

Proposed Solution. Aspect-oriented programming (AOP) [2] enables developers to capture crosscutting concerns that spread over different components of a system. It provides an approving solution to crosscutting concerns by introducing a new program construct, the aspect, which is used to localize and encapsulate crosscutting concerns within separate modular units and is also used to describe how these units integrate with primary parts of the system. The AOP paradigm provides a more clear and understandable system. In order to improve the modularity and maintainability of scientific computing programs, we introduce aspect-oriented techniques to treat error handling concerns as aspects. In this paper, a case study is carried out on the SOF system to empirically investigate whether AOP can help implement and maintain the error handling policies of the program under study. We first distill error handling policies to the system, and then refactor them as aspects with AspectJ. Finally, we evaluate the feasibility of our proposal by comparing the AspectJ version with the original implementation.

Contributions. The main contributions of this paper is to apply the aspect-oriented techniques to separate computational error handling concerns from scientific computing programs into aspects. We conduct a case study on a real-world scientific computing system SOF, and evaluate the improvements and pitfalls of the aspect-oriented implementation with a series of experiments.

The rest of the paper is organized as follows. We start with an introduction of the background on AOP and computational errors in section 2. After that, in section 3, we refactor computational error handling concerns in the SOF
2. BACKGROUND

2.1 Aspect-Oriented Programming

A crosscutting concern is a special concern that spread over different components of a system and overlaps multiple other stakeholder interests. The design and implementation of this kind of concern are critical problems during software development process. Traditional solutions most focus on vertically decomposing and encapsulating concerns into individual modules. However, they cannot localize such concerns that cut across more than one primary functional features to a single module.

Gregor Kiczales and his colleagues originated the concept of AOP [2]. AOP complements OOP by allowing developers to capture and localize crosscutting concerns into aspects. This mechanism allows crosscutting concerns to be implemented in a more modular way, which achieves the usual benefits of improving modularity: simple code which can be understood and maintained more easily, closer to the human perception, and greater ability of changing and reusing to meet new requirements.

2.2 Computational Errors

Computational errors in scientific computing can be categorized into model error, observation error, truncation error, and rounding error. Firstly, mathematical models for realistic problems are constructed by abstracting and predicting, so the solution is approximate. This kind of error is called model error. Furthermore, there are some physical quantities in a mathematical model, such as: temperature, speed, pressure and so on. These quantities are observed from measuring equipments, which would generate observation errors. In addition, the numerical method is employed to work out approximate solutions for mathematical models. The error between the exact solution and the approximate solution is called truncation error or method error. Finally, computers use fixed length of bits to store float numbers, and intermediate computing results are approximated by assigned rounding rules. The error caused by rounding is called rounding error. Because errors are unavoidably introduced during the process of solving mathematical problems with numerical methods, error handling are absolutely necessary for figuring out solutions that satisfy required precision. In this paper, we only focus on the handling of truncation errors.

Although there is a generic need of error handling for scientific computing software, yet the demanded precision of computing results is changeful according to different objectives and environments. To meet specific computing precision requirements, computing parameters even algorithms need to be optimized or adopted. The principal difficulty of dealing with errors in scientific computing software is that error handling is a system-wide concern that spread over all different modules of the software. Typically, these codes are scattered over the whole system and tangled with the primary functional codes. Hence in current practices, it is very often to find and modify codes which implemented error-related concerns, to meet changing requirements. As a result, the traditional way is not only error-prone but also makes it difficult to verify the correctness of the implementation and to perform maintenance tasks.

3. ERROR HANDLING AS ASPECTS

3.1 The Satellite Orbit Forecast System

The SOF system is a real-life scientific computing system developed to forecast the orbits of earth satellites by ground observation stations. By inputting initial coordinates of an orbit and related parameters, the system will compute the successive coordinates of the orbit and synthesizes the median orbit of the satellite during a fixed period of time. The core module of the system is a set of integrators which includes RKF4(5), RKF5(6), RK6(7), and RKF7(8) [3]. These integrators are used to figure out the differential equation of satellite orbits. The orbit differential equation is in the form of: \( y'(t) = f(t, y(t)) \), in which \( y \) is a vector of the orbit’s coordinates, \( t \) is the time. The numerical solutions of the equation represent the expected coordinates of the orbit in a specific time.

Given a time range \([a, b]\) and an initial value \( y(a) = y_a \), to solve the equation with numerical method is to figure out \( m \) approximate coordinate vectors \( y(t_n) \) for \( y_a \) step by step (\( a = t_0 < t_1 < t_2 < ... t_n < ... t_m = b \)). In order to solve \( y_n \), a numerical integrator takes a step size \( h_{n-1} \) in the direction of \( b \), \( t_n = t_{n-1} + h_{n-1} \), then compute \( y(t_n) \), which is an approximate solution of \( y_n \). The parameters and integrators need to be adjusted according to different needs of computing time and precision. The appropriate size of integration step for different shapes of orbits is not the same, and the appropriate integration algorithm is also varied for different satellites. So the system has exigent requirements for flexibility and changeability.

Originally, this system is implemented with Java which is one of the efficient languages to develop scientific computing programs [4]. The original implementation consists of 2638 lines of code, and 250 lines are related to error handling. The codes related to error handling cut across the whole system for 15 places. As a result, any change of error handling policies will inevitably lead to perform a large amount of modifications sprinkling throughout the whole system. And maintenance tasks cannot be done without modifying the modules of primary functionalities. This way of maintenance could introduce unpredictable exceptions. Facing this problem, we plan to utilize aspect-oriented techniques to encapsulate error handling polices into aspects to improve the modularity and maintainability of the system.

3.2 Aspects in the SOF system

In order to improve the modularity and changeability of the SOF system, we refactor 4 aspects of error handling polices. They are descent, error estimation, step size adaption, and algorithm adaption aspects. At runtime, these aspects will executed to optimize, estimate, and control computational errors under the required computing accuracy without any intervention at runtime. Future changes of error handling polices can also be effectively realized by aspect orientation mechanisms without affecting codes of primary functionalities.

3.2.1 Error Optimization Aspect

In practice, scientific computing problems are often nonlinear, strong restricted, stochastic, and large-scaled. Simple
and single algorithms may cause potential problems such as local minimum and divergence. So, combining simple algorithms with proper optimization algorithms is absolutely necessary to reduce errors under an accepted level. Some optimization algorithms are used to solve a general class of computation problems by combining user-generated black-box procedures. For instance, Chaotic and Simulated Annealing methods are in common used to avoid local minimum of objective functions. These optimization algorithms are not closely related to concrete problems, and should not be tangled with the concrete computing problems. We intend to encapsulate optimization strategies as aspects.

In the case study, the Kepler equation, which in the form of \( E - M = e \cdot \sin(M) \), is used to compute the ephemeris of satellites, the moon, and the solar. Newton method is used to figure out the variable \( E \) in the Kepler equation with known \( e \) and \( M \). The linear procedure of Newton method can be described as: compute \( f(x_n) \) with an iterative formula \( x_{n+1} = x_n - f(x_n) / f'(x_n) \) until \( f(x_n) \) is smaller than a threshold value \( \varepsilon \). In the formula, \( f(x_n) \) is used to compute the result of the equation and \( f'(x_n) \) is used to compute the result of the equation’s derivative. The algorithm of Newton method is described in List 1.

List 1. Newton method for differential equations

```c
do
  x_n = x_{n+1}
  f_n = f(x_n) // f() computes the result of the equation
  f'_n = f'(x_n) // f'(x_n) computes the result of the equation’s derivative
  x_{n+1} = x_n - f_n / f'_n
while(|f_n| > \varepsilon)
return x_{n+1}
```

The Newton method is employed to solve the differential equation because the iteration process can end in a converged solution rapidly. However, the convergence domain of Newton method is relatively narrow. In other words, this method can only guarantee to be converged when initial values are close to the exact solution. As an enforcement, the descend method is used to make sure intermediate results of the objective function are descended stably to the exact solution after each iteration. When \( f(x_{n+1}) > f(x_n) \), the descend algorithm will revise \( x_{n+1} \) to \( \lambda \cdot x_{n+1} + (1 - \lambda) \cdot x_n \).

The combination of Newton method and descend method guarantees the algorithm to converge in a wider domain of initial values. The descend method plays as an optimization algorithm for Newton method. Separating the descend algorithm with the concrete equation solving process makes the code more comprehensible. Furthermore, the descend method is more amenable to be reused as a general optimization strategy. The algorithm of descend is described as an aspect in List 2.

List 2. Descend optimization for Newton method

```c
pointcut Descend() execution Newton (...) & & set x_n+1
after (_) : Descend (){
  if(|f'(f(x_n))| > |f(x_n)|)
    x_n+1 := \lambda \cdot x_{n+1} + (1 - \lambda) \cdot x_n
}
```

3.2.2 Error Estimation Aspect

Error estimation is to compute an approximate value of the exact error, just as the primary solution is an approximate value of the exact solution. The most important reason for error estimation is to archive confidence of the primary solution. Although error estimation will inevitably bring in additional costs, it is indispensable generally, because the solutions without precision confidence are meaningless. In addition, the estimated error can be used to control errors. Error estimation of integrators should take both local error and global error into consideration. Local error is the error produced in per unit of integration step, while global error is the error accumulated during the whole integration process.

Embed Runge-Kutta method is used in this case study to solve the differential equation of orbits [3]. The basic idea is to construct pairs of self-contained Runge-Kutta formulas. In addition to figure out an approximate solution of \( f(x_n) \) with the order formula, an approximate solution of \( f^*(x_n) \) is also figured out with the order formula. Then \( |f^*(x_n) - f(x_n)| \) is an estimation of the local error for \( f(x_n) \) which is computed with the lower order formula. The \( p \) and \( p + 1 \) order formulas are constructed sophisticatedly. In order to figure \( p + 1 \) order formula, only a few more calculating times are needed to compute the right function after the \( p \) order formula is solved. In this case study, the local error is estimated by using the formula given in [3].

An usual way to estimate global error is integrating by two different order formulas with same step size. A basic approximate \( f(b) \) of order \( p \) and another approximate \( f^*(b) \) of order \( p^* \) are both computed, in which \( p^* > p \). Then \( |f^*(b) - f(b)| \) is an asymptotically estimation of the global error for \( f(b) \). Another method to estimate global error is integrating with two different step sizes simultaneously. A basic approximate \( f(b) \) of step \( h \) and another approximate \( f^*(b) \) of step \( h^* \) are both computed, in which \( h^* < h \). Then \( |f^*(b) - f(b)| \) is the estimated global error of \( f(b) \). The algorithm of global error estimation by integrating with two different step sizes is encapsulated as an aspect in List 3.

List 3. Global error estimation aspect

```c
pointcut GlobalErrorEstimate() call
  Integrate(double y[], double h) / y is the vector of orbit coordinates, h is the size of integration step
after (_ ) : GlobalErrorEstimate(){
  // y* is the vector of orbit coordinates integrated with h/2
  y* = Integrate(y, h/2)
  y = Integrate(y, h)
  double E = 0, y = the global error estimated
  for each yi in y*
    E += |yi - y[i]|
  if(E > MaxE)
    // record maximal error during integration process
    MaxE = E
  else if(E < MinE)
    // record minimal error during integration process
    MinE = E
}
```

3.2.3 Error Controlling Aspects

Appropriate parameters and algorithms are two key issues to obtain numerical solutions that satisfy required calculation precision with acceptable performance costs. It is not practical to solve real-life numerical computing problems with fixed parameters and algorithms, because some variables can only be confirmed at run time. For example, in an integration program, the deflection rate of the integral curve cannot be known before the integration is finished, but the suitable integration step size and algorithm are determined by the deflection rate.

In the case study, the deflection rate of a differential equation integral curve changes significantly in a given integration scale. It is improper to solve orbit equations with a fixed integrator and a constant step size. In one hand, resolving the fastest change of the the integral curve by integrating with small enough step size or high enough order formula to
meet the required precision will lead to too much computing costs. In another hand, taking affordable performance costs into account, integrating with big enough step size or low enough order formula could not meet the required computing precision. There are two most common ways used to cope with the dilemma of accuracy required and performance costs in solving differential equations. One is controlling errors by adjusting step sizes; the other is controlling errors by adjusting integrators. Both of the two methods can stabilize the integration and balance the required precision and the performance costs.

### Step Size Adaption Aspect

There are two major factors affecting the choice of step size. One is the accuracy, and the other is the accumulating error. If the step size is too big, the computation could become unstable. Conversely, when the step size is too small, the integration will require more time to finish the integral domain. This will bring in bigger accumulate errors as well as more computing costs. As a result, it is necessary to adjust step size to solve a differential equation, especially when the order of the determinant is relatively large. Step size adaption can stabilize the integration process and control errors below an acceptable level. Although there are some costs associated with error estimation and step size adaption, this is a bargain generally because the differential equation is solved more efficiently. Step size adaption is an independent error control policy which should not be tangled with the primary process. We implement it in an aspect in List 4.

**List 4. Step size adaption aspect**

```java
@pointcut StepSizeAdaption() call RKF*(..)
@after () : StepSizeAdaption {()
  if(E < MinET) /*error estimated is smaller than the lower bound of error threshold*/
    h=h*2 // adjust step size to a bigger value
  else if(E > MaxET) /*error estimated is bigger than the higher bound of error threshold*/
    h=h/2 // adjust step size to a smaller value
  redoRKF*(..)
}
```

### Integrator Adaption Aspect

Integrating with a lower order integrator could be faster but less accurate, otherwise, solving the differential equation with a higher order integrator is more accurate but could introduce unaffordable performance costs. There is a tradeoff to choose an appropriate order of integrator. Indeed, if the integral curve changes significantly in an integration scale, it could cost too much to resolve the fastest changes of the integral curve with a high order integrator. On the contrary, if the order of the integrator is too low, the precision of the computing result could become unacceptable. Adjusting integrator is another way to stabilize the integration process and control errors below an acceptable level. Integrator adaption is also an independent error control policy which should not be tangled with the primary process. We implement it in an aspect in List 5.

**List 5. Integrator adaption aspect**

```java
@pointcut IntegratorAdaption() call RKF*(..)
@after () : IntegratorAdaption {()
  if(E < MinET) /*error estimated is smaller than the lower bound of error threshold*/
    set current integrator to lower order integrator
  else if(E > MaxET) /*error estimated is bigger than the higher bound of error threshold*/
    set current integrator to higher order integrator
  redoRKF*(..)
}
```

### EVALUATION AND DISCUSSION

The technologies used in the implementation of the case study are JDK 1.6.0, AspectJ 1.5.2, and Eclipse 3.3.2 under window XP environment, and the experiments are carried on a computer with P4 2.93 GHz processor and 2GB memory.

The 4 computational error handling concerns are implemented in two distinct ways: originally inject in the program with standard Java; and refactor as aspects with AspectJ. The functionalities of the two implementations are exactly the same. Walker et al. [5] used the time which is spent on debugging and modifying as metrics to evaluate the effectiveness of AOP. However, the value of this metrics depends on respondents’ experience on AOP and OOP. Therefore we selected more objective metrics, such as lines of code (LOC) and the amount of affected places, to evaluate the modularity and maintainability. The two versions of implementation are compared in table 1. The results show that AOP has better modularity in dealing with error-related concerns of scientific computing system, because less LOC and places are related to error handling in the AspectJ version.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>LOC</th>
<th>Places Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Java</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>AspectJ</td>
<td>55</td>
<td>1</td>
</tr>
</tbody>
</table>

In addition, we compare the efforts which need to be performed when we manage to change error-related policies in both of the two versions of the SOF system after implementation finished. The results in table 2 show that when changing error-related concerns, fewer lines of codes and places are affected in the AspectJ version. This comparison illustrates the maintainability improvement of using AspectJ to treat error-related problems in scientific computing programs as aspects. Finally, we compare the execution times of the two versions of implementation in table 3. The results show that there is no noticeable difference in performance between the two versions of implementation.

To sum up, the benefits of implementing computational error handling concerns as aspects are as follows. First, the code looks much cleaner and shorter in the AspectJ version, which makes the implementation easier to be understood. Second, now it is feasible to implement a simple primary function first, and then add error handling policies into it without affecting the primary function modules. Separating error-related concerns into aspects makes the program easier to be adapted and changed than using conventional Java. Furthermore, the separation of concerns allows developers to focus only on the numerical computing task of the primary problem. As a result, significant improvements in terms of modularity and code reusability can be obtained. Besides, the experiments also show that the performance of the SOF system doesn’t decrease notably in the AspectJ version. In addition, AspectJ also facilitates the debug process, because it provides a mechanism of excluding and including error optimization, estimation, and control aspects of the program to compile and execute conveniently.

The results of the case study favor the aspect-oriented paradigm mostly. We can arrive to the conclusion that AOP is an adaptive method to deal with error handling concerns. Introducing AOP into scientific computing software development can increase modularity and decrease maintenance...
costs. However, we also found some issues when implementing the AspectJ version of the SOF system.

- Before advice for set pointcut. Semantically, a before advice for a set pointcut can extract the previous value of a variable before an assignment. Whereas, it returns the value after the assignment. We have to use the reflection mechanism of Java to acquire the value of the variable before the assignment.
- Statement level pointcut. Because AspectJ is lack of statement level pointcut support, we add hook methods [6] with empty bodies as hooks around statements. If statement level pointcuts are supported, the hook methods can be removed. This will make the code looks more simple and natural.
- Local variables access. Scientific computing programs are data centric in a large extent, so monitoring variables in some methods is necessary. Local variables in a method are not visible for aspects in the current AspectJ. The developing of scientific computing software with AspectJ would be leveraged by providing the mechanism of accessing local variables in aspects.

5. RELATED WORK

Solving mathematical problems with numerical analysis will inevitably bring in computational errors. The most important reason for handling errors is to gain some confidence in the numerical solution. Shampine et al. propose algorithm to estimate control errors. In [7], the error handling concerns are tangled with the primary initial value problem of ordinary differential equations.

In order to assess the effectiveness of AOP, there are many case studies on how to apply AOP for different crosscutting properties of system, like security [8], persistence [9], multi-thread interactions [10], etc. Harbulot et al. Introduce AOP into the realm of scientific computing, but the focus on using AspectJ to encapsulate parallelization of scientific java code into aspects[11]. While in this paper, we try to separate computational error handling concerns as aspects.

The errors handled with aspects in [12] are program exceptions, whereas we deal with error optimization, estimation, and control concern as aspects in scientific computing software. To the best of our knowledge, this paper is the first attempt to introduce aspect-oriented mechanisms to treat computational error handling concerns as aspects.

6. CONCLUSIONS AND FUTURE WORK

Traditional error handling approaches result tangled code which is hard to be understood and adapted. In this paper, we propose to encapsulate error handling concerns as aspects, and present a case study on a real-world scientific computing system SOF to evaluate our approach. By refactoring error handling concerns as aspects, the AOP version improves the modularity and maintainability of the system without noticeable compromise in performance.

In the future, we plan to continue our work based on this case study, and try to find out whether it is practical to use aspect-oriented techniques to solve error handling polices in more detail levels of scientific computing programs.

7. ACKNOWLEDGEMENT

The authors are supported by the National Natural Science Foundation of China (No.60721002, No.90818022, and No.60603036), the Jiangsu Province Research Foundation (BK2007139), and Scientific Research Foundation of Graduate School of Nanjing University (No.2008CL07). The authors would like to thank Dr. Xi-Yun Hou for his helpful discussions.

8. REFERENCES