Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 126069, 7 pages
http://dx.doi.org/10.1155/2015/126069

Research Article

Emulating Multiple Inheritance in Fortran 2003/2008

Karla Morris

Sandia National Laboratories, 7011 East Avenue, Livermore, CA 94550-9610, USA

Correspondence should be addressed to Karla Morris; knmorri@sandia.gov

Received 24 July 2012; Accepted 24 January 2015

Academic Editor: Can Özturan

Although the high-performance computing (HPC) community increasingly embraces object-oriented programming (OOP), most HPC OOP projects employ the C++ programming language. Until recently, Fortran programmers interested in mining the benefits of OOP had to emulate OOP in Fortran 90/95. The advent of widespread compiler support for Fortran 2003 now facilitates explicitly constructing object-oriented class hierarchies via inheritance and leveraging related class behaviors such as dynamic polymorphism. Although C++ allows a class to inherit from multiple parent classes, Fortran and several other OOP languages restrict or prohibit explicit multiple inheritance relationships in order to circumvent several pitfalls associated with them. Nonetheless, what appears as an intrinsic feature in one language can be modeled as a user-constructed design pattern in another language. The present paper demonstrates how to apply the facade structural design pattern to support a multiple inheritance class relationship in Fortran 2003. The design unleashes the power of the associated class relationships for modeling complicated data structures yet avoids the ambiguities that plague some multiple inheritance scenarios.

1. Introduction

Object-oriented programming originated in the late 1960s with the invention of the Simula 67 computer programming language. Since the early 1980s several modern programming languages, including C++, Java, Python, and Ruby, have been developed with explicit support for OOP from their inception. Several older languages, including Ada, BASIC, Lisp, Pascal, and Fortran, have evolved to support OOP in their modern versions [1]. Of these two groups, Fortran and C++ find the most frequent use in high-performance computing [2]. Due to the longer history of C++ support for OOP, a much more extensive body of literature exists for describing best practices in C++. The recent advent of widespread compiler support for the OOP features of the Fortran 2003 and 2008 standards offers an exciting opportunity to develop and promulgate idioms for expressing OOP concepts in Fortran [3]. The current article focuses on one such idiom: multiple inheritance.

The primary unit of decomposition in OOP is the object. The early stages of any object-oriented software design process involve defining classes of objects encapsulating the problem data and methods that manipulate the encapsulated data. The methods, referred to as *type-bound procedures* (TBP) in Fortran, provide functionality to the objects and the objects' users [4]. The objects can be organized into class hierarchies to define complex data relationships. One important feature most OOP language provide is dynamic polymorphism: the ability of an object at one level in the hierarchy to respond to invocations of methods inherited from one of the object's ancestors in the hierarchy.

Numerous designs can accomplish a desired task, but certain recurring design problems arise. In object-oriented design, the set of common solutions for recurring problems are termed *design patterns*. Designs that express these patterns typically exhibit higher quality in various respects [5]. Experienced software developers leverage design patterns to implement class hierarchies that have proven effective in solving a specific design problem across a variety of application domains. HPC projects that employ patterns include the open-source parallel solver libraries Trilinos (http://trilinos.sandia.gov/) and Parallel Sparse Basic Linear Algebra Subroutines (PSBLAS) (http://www.ce.uniroma2.it/psblas/).

Design patterns imbue several desirable properties in software: flexibility, maintainability, reusability, and scalability. As an example, consider the Mediator pattern. Common design problems involving a fully connected graph

connecting interacting objects can be reduced to simple star graph by the Mediator pattern [6]. A Mediator is a single object that directs communication between N other objects, thereby reducing the communication links from $O(N^2)$ to $O(N)$. The resultant reduction in complexity increases the maintainability of the code and the scalability of the design up to more classes than would otherwise prove feasible. PSBLAS, for example, defines different compressed storage schemes for its sparse matrix objects. Because the scalability of certain operations is improved by using specific storage representations, there is functionality within the library to transform the storage scheme of its objects. The library uses the Mediator pattern to manage the number of procedures that must be implemented to convert the compressed storage representation of sparse matrix objects [7].

Interface reusability is another desirable software property. The Factory Method pattern defines a unifying interface for constructing objects across different classes that support the unified construction interface. Xpetra, a package in Trilinos, uses the Factory Method pattern to defer the instantiation of actual objects to an appropriate subclass implemented in other packages.

Inheritance addresses another important concern in software design: implementation reuse. Inheritance involves constructing child classes that automatically acquire attributes and methods of a parent class. One form of inheritance, multiple inheritance, incorporates functionality from more than one ancestor class.

Multiple inheritance also proves useful in applications where a programmer prefers class relationships structured into a lattice rather than a strict tree hierarchy class relationship structure [8]. Lattices and multiple inheritance find specific use in separating an application's interface from its implementation [9].

OOP language designers have chosen different levels of support for multiple inheritance. Each language that explicitly supports multiple inheritance must resolve ambiguities that might arise. The diamond problem, a common ancestor delegation dilemma problem, is an example of an ambiguity that arises when two classes B and C inherit from a class A. If a class D in turn inherits from B and C, then any method in A overridden with implementations in B and C but not in D has two alternative and ambiguous implementations when invoked on D: the implementation from parent B or that from parent C (if the method is not overwritten by D) [10].

Several OOP languages fully support multiple inheritance. These languages include C++, Python, Perl, and Curl. C++ partially resolves ambiguities such as those previously cited by enforcing explicit qualification, as each path in the inheritance structure is followed separately [8]. Java, Ruby, and C# limit support to inheritance from only one class implementation; however, a class can implement multiple interfaces (Java terminology), which would correspond to extending multiple abstract classes in Fortran terminology, but Fortran prohibits even this limited form of multiple inheritance.

What appears as an intrinsic feature of one language, however, might appear as a user-defined design pattern in another language. In this sense, design patterns are programming-language-independent. Only the degree of language support for the pattern varies. The Prototype pattern [6], for example, is a common user-defined pattern in C++ that is an intrinsic language feature in Fortran 2008 [7].

Publishing techniques for emulating unsupported aspects of OOP in Fortran has a long history. Starting in the mid-1990s, several groups published strategies for emulating OOP in Fortran 90/95 [11–15]. Scientific programming developers have also implemented design patterns in Fortran 90/95 before object-oriented compiler support [16]. No publication of which the current author is aware addresses emulating multiple inheritance in Fortran. Section 2 of this paper describes a multiple inheritance pattern using Unified Modeling Language (UML) diagrams [17]. Section 3 presents a sample implementation of the pattern leveraging the OOP features of the Fortran 2003 standard. The latest released versions of at least six Fortran compilers support all of the OOP features employed in Section 3: the IBM, Intel, Cray, Numerical Algorithms Group (NAG), Portland Group, and GNU Compiler Collection (GCC) compilers [3]. Section 4 discusses the results. Section 5 concludes the paper.

2. Methodology: Multiple Inheritance Pattern

The model presented in this section applies a Facade software design pattern to support the implementation of a multiple inheritance pattern in Fortran 2003. In general, the Facade structural design pattern can be used to reduce the complexity of a system and decouple multiple dependencies between subsystems. The client interacts directly with the facade, which provides a simple and unified interface to a set of subsystems [6]. In following a format similar to that prescribed by Gamma et al., this section describes the proposed multiple inheritance design pattern by using a set of key features: intent, motivation, applicability, participants, collaborators, and implementation.

(i) *Intent*: create a superclass that inherits functionality from a set of subclasses. Support class modularity by allowing a program structure where the capabilities of a superclass result from inheriting the features that are implemented by different subclasses.

(ii) *Motivation*: the collection of capabilities of a system is better implemented by using a structure set of subsystems. Each subsystem is responsible for implementing a specific capability. This module structure contributes to data and functionality encapsulation and enables the reusability of the different subsystems. The complexity of the system is managed by minimizing the communication and dependencies between the different components in the system.

Consider for example the basic constructs involved in a linear algebra application such as vectors, scalars, and matrices. In a parallel programming environment these constructs could take the form of superclasses that encapsulate distributed data, and basic functionality for data redistribution and communication, as

well as functionality for basic computations, floating-point operations, and memory management, among others. If each of the different functionalities previously mentioned is implemented separately both the vector and matrix classes can reuse each of the classes responsible for the implementation of their common capabilities.

(iii) *Applicability*: the multiple inheritance pattern is used when a newly created class, which we refer to as a superclass, inherits the properties and behavior of multiple classes. The superclass delegates to its appropriate subclass any request for functionality implemented within the subclass, whether the request is generated inside the superclass or by an outside client using the superclass.

(iv) *Participants and Collaborators*: Figure 1 shows a UML class diagram of the multiple inheritance pattern. A similar data structure is commonly used within the Epetra (http://trilinos.sandia.gov/packages/epetra/) foundational package of the Trilinos library. In this case, the classes `object_distribution` and `object_computation` are part of the subsystems responsible for implementing the specific features related to data distribution and computation of floating-point operations, respectively.

The `object_distribution` class is one of the classes in a subsystem that implements all the functionality required to address data redistribution and enable import and export capabilities. The class `object_computation` is included in a subsystem that computes and reports computation of floating-point operations. The system, embodied by the `vector` class, makes use of the collection of capabilities.

The `vector` class could modify the implementation of any TBP of the two inherited classes or implement new functionality using both inherited classes. In an OOP language that explicitly supports multiple inheritance, the `vector` class inherits from both `object_distribution` and `object_computation`. All procedures attached to these classes thereby would be directly available to `vector` objects.

(v) *Implementation*: the proposed multiple inheritance pattern emulates the language-provided multiple inheritance of C++ and can be viewed as an application of the Facade pattern. Figure 2 provides a UML class diagram describing the proposed multiple inheritance pattern.

The model embeds a Facade pattern to collect all the classes that are to be inherited by the `vector` class. The created lattice structure comprises three classes: `object_distribution`, `object_computation`, and `facade`. The `facade` class combines the functionality provided by two demonstrative classes: `object_distribution` and `object_computation`.

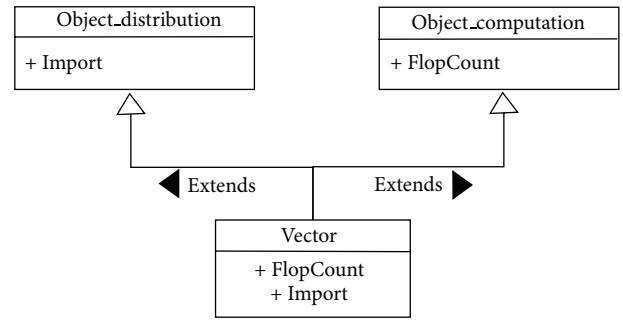

Figure 1: UML class diagram for multiple inheritance pattern, which is provided by the compiler in languages like C++. The vector class inherits from two other classes: object_distribution and object_computation.

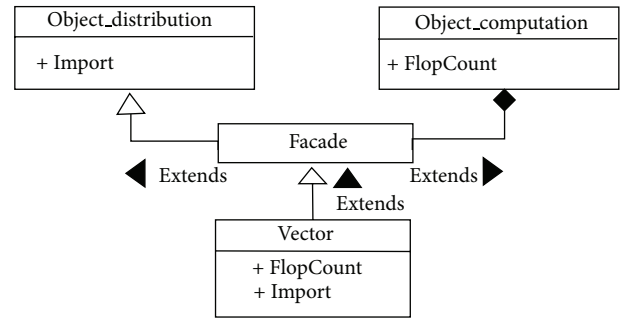

Figure 2: UML class diagram for proposed multiple inheritance pattern Fortran implementation. The facade class aggregates the functionality encapsulated in object_distribution and object_computation and provides a unified interface for the classes the vector class inherits.

The `vector` class shown could be any class that must extend the collection of classes handled by `facade`.

3. Results: Fortran Implementation

The Fortran implementation for emulating multiple inheritance with the previously described model is shown in Listings 1–4. Multiple inheritance is applied to enhance the number of capabilities available to a superclass. The design uses a Facade to collect all inherited functionality from a set of subclasses that form the lattice data structure. In this example, the data structure encapsulates functionality related to data distribution and computation capabilities.

As part of a subsystem that enables data distribution the `object_distribution` class, in Listing 1, includes the `import` method. This type-bound procedure allows `vector` to change the layout of its data in memory. The `object_computation` class, shown in Listing 2, encapsulates floating point operation functionality and allows `vector` objects access to the floating-points operations performed by the superclass.

```
(1) module object_distribution_concrete
(2)   implicit none
(3)   private
(4)   public:: object_distribution
(5)   type:: object_distribution
(6)     contains
(7)       procedure:: import
(8)   end type
(9)    contains
(10)     subroutine import(this)
(11)       class(object_distribution), intent(in):: this
(12)       print *, 'Import␣Data␣Functionality'
(13)     end subroutine
(14) end module
```

Listing 1: Implementation for object_distribution class.

```
(1) module object_computation_concrete
(2)   implicit none
(3)   private
(4)   public:: object_computation
(5)   type:: object_computation
(6)     contains
(7)       procedure:: FlopCount
(8)   end type
(9)    contains
(10)     subroutine FlopCount(this)
(11)       class (object_computation), intent(in):: this
(12)       print *, 'Count␣floating␣point␣operations'
(13)     end subroutine
(14) end module
```

Listing 2: Implementation for object_computation class.

The source code in Listing 1 illustrates several OOP features, including data encapsulation support. In Fortran, the `private` and `public` keywords (lines 3-4) control access to members of the type. Members defined with `public` are accessible to any part of the program. Conversely, members defined with `private` are not accessible to code outside the scope of the module in which the type is defined. The `implicit none` statement on line 2 prevents implicit variable instantiation and forces programmers to declare all variables before they are used.

The `facade` class collects the functionality of the subclasses through inheritance and composition (see Listing 3). The `constructor` function (lines 16–19) returns a facade object after invoking intrinsic structure constructors for each data member, my_flop, and the extended `object_distribution` class. Fortran automatically provides an intrinsic structure constructor that returns an object wherein any private data components inside the object take on programmer-defined default initializations. Public state variables can be initialized by passing arguments to the structure constructor or by overloading the derived type name and providing a programmer-defined constructor function. The later is a more desired approach when following an object-oriented programming paradigm, which requires private state variables for the derived type data components. In this code example, (lines 17-18) construction relies upon default initializations and therefore requires no arguments. Through inheritance, `object_distribution` gives `facade` objects, and any of its extended classes, full access to `object_distributions`'s TBP. The composition relationship use for `object_computation` forces `facade` to create a TBP, `FlopCount`. This procedure invokes the implementation of FlopCount associated with the `facade` my_flop component.

The `Vector` superclass, shown in Listing 4, represents any derived type that accesses, through the inheritance of `facade`, all the collective functionality from the subclasses in the lattice structure.

The design described aggregates within `facade` all the classes that must be inherited by the `vector` derived data

```
(1) module facade_concrete
(2)   use object_distribution_concrete, only: object_distribution
(3)   use object_computation_concrete, only: object_computation
(4)   implicit none
(5)   private
(6)   public :: facade
(7)   type, extends(object_distribution):: facade
(8)     type(object_computation):: my_flop
(9)     contains
(10)       procedure:: FlopCount
(11)   end type
(12)   interface facade
(13)      module procedure constructor
(14)   end interface
(15)   contains
(16)      type(facade) function constructor()
(17)        constructor%object_distribution=object_distribution()
(18)        constructor%my_flop=object_computation()
(19)      end function
(20)      subroutine FlopCount(this)
(21)        class(facade), intent(in):: this
(22)        call this%my_flop%FlopCount()
(23)      end subroutine
(24) end module
```

Listing 3: Additional level of abstraction used to incorporate all classes to be inherited.

```
(1) module foo_concrete
(2)   use facade_concrete, only: facade
(3)   implicit none
(4)   private
(5)   public:: foo
(6)   type, extends(facade):: foo
(7)   end type
(8)   interface foo
(9)    module procedure constructor
(10)  end interface
(11)   contains
(12)     type(foo) function constructor()
(13)       constructor%facade=facade()
(14)     end function
(15) end module
```

Listing 4: Vector class taking advantage of multiple inheritance.

type. The `facade` class adds a level of indirection to encapsulate the multiple classes that are to be inherited. The type-bound procedures in the `facade` class provide `Vector` with direct access to the implementations in the parent classes in Listings 1 and 2.

The aforementioned diamond problem could occur when implementing the UML diagram shown in Figure 3. The `object` class holds some common, high-level, required functionality such as memory management. As such, any derived data type that implements a new capability must extend the `object` class. This behavior is enforced through an Object pattern and has been used in a reference-counting package [18, 19]. The Object pattern named after a like-named feature in Java propagates common functionality throughout a class hierarchy by establishing a base class that serves as a universal ancestor of all classes in a given package [20, 21]. If an abstract class defines the universal parent `object`, a designer could employ deferred bindings, which comprise type-bound procedures with only their function signatures specified. Writing deferred bindings provides a hook for accessing a particular capability in all concrete classes further down the hierarchy. Another approach provides a default implementation within the parent class; in this case, subclasses have the option to overwrite the procedure unless the deferred binding has the `non_overridable` attribute.

In an application where the `object` class publishes an interface for a TBP `free_memory`, the diamond problem will arise if concrete implementations are furnished by the concrete classes `object_distribution` and `object_computation`. Given that the `vector` class extends both concrete classes, invoking the `free_memory` method poses a dilemma: it is unclear which concrete implementation should be used. The pattern implementation shown in Figure 2 avoids this common ancestor dilemma; in this case, if we were to invoke the `free_memory` method, the implementation used would be given by `object_distribution` class with no ambiguity.

The pattern implemented in Listings 1–4 can also address the needs of a software application where the desired behavior requires the use of both concrete implementations of

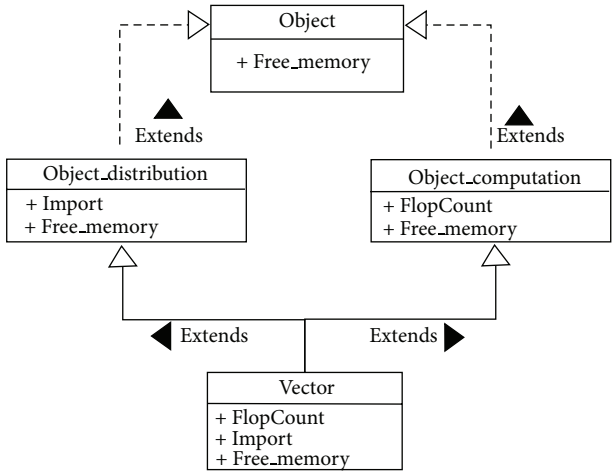

Figure 3: Unified modeling language (UML) class diagram for multiple inheritance diamond problem, given an interface for type-bound procedure free_memory.

the `free_memory` type-bound procedure. In such cases, the procedure should be overwritten within the `object` class, to invoke the implementations of the different parent classes. If the user code requires knowledge of the parent TBP invoked by `facade`, that could be discovered at runtime with Fortran's `extends_type_of` intrinsic procedure.

4. Discussion

The multiple inheritance pattern implementation discussed in Section 3 requires little work to incorporate additional functionality into the lattice structure. The `facade` used to emulate multiple inheritance with this pattern can be better exploited when the application requires that this collective functionality be used by several different superclasses. New functionality can be added to all the objects in the application, or all superclasses of interest, by specifying a new component in the `facade` class, corresponding to the derived type of the new features.

In the model described, the inheritance of multiple classes is achieved by inheriting one of the classes and aggregating the rest of the classes in the set providing the collective functionality. In the example, the set is given by classes at the hierarchy level of `object_distribution` and `object_computation`. The class that should be inherited by the `facade` derived data type, which encapsulates the multiple inheritance, is given by the class with the procedure implementation that would generally satisfy the inheritance path appropriate for the specific application.

In applications where all inheritance paths must be followed, the `free_memory` type-bound procedure should be overwritten within the `facade` class. Its new implementation will invoke the concrete implementation of the method for each one of the classes providing the collective functionality. Overwriting methods within this pattern's `facade` class affords the programmer the flexibility of implementing the desired behavior and at the same time circumvents the occurrence of possible ambiguities in the subclasses. A multiple inheritance design pattern implementation solely based on aggregation could prove valuable if the collective functionality requires a complex interaction between the different methods provided by the subclasses.

5. Conclusion

The discussed model presents a safe solution for emulating multiple inheritance in Fortran 2003/2008. The pattern avoids the ambiguities that commonly arise when dealing with lattice data structures and at the same time provides the means for supporting different implementations. Different behaviors within a given software application can be supported depending on the inheritance path that needs to be followed. The pattern can successfully address the needs of applications where either a single concrete implementation of a method is required or applications where the implementation of the methods for each of the subclasses in the structure must be invoked.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Sandia National Laboratories is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy under Contract DE-AC04-94AL85000. The author thanks Damian Rouson for the helpful discussions and guidance.

References

[1] G. O'Regan, *A Brief History of Computing*, Springer, London, UK, 2nd edition, 2011.

[2] M. Bull, X. Guo, and I. Liabotis, "Applications and user requirements for tier-0 systems," Tech. Rep. INFRA-2010-2.3.1, Partnership for Advanced Computing in Europe (PRACE) Consortium Partners, 2011.

[3] I. D. Chivers and J. Sleightholme, "Compiler support for the Fortran 2003 and 2008 standard," *ACM SIGPLAN Fortran Forum*, vol. 31, no. 1, pp. 23–33, 2012.

[4] M. Metcalf, J. K. Reid, and M. Cohen, *Fortran 95/2003 Explained*, Oxford University Press, Oxford, UK, 2004.

[5] A. Shalloway and J. R. Trott, *Design Patterns Explained*, Addison-Wesley, Reading, Mass, USA, 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, New York, NY, USA, 1995.

[7] D. Barbieri, V. Cardellini, S. Filippone, and D. Rouson, "Design patterns for scientific computations on sparse matrices," in *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *Lecture Notes in Computer Science*, pp. 367–376, Springer, 2012.

[8] B. Stroustrup, "Multiple inheritance for C++," in *European UNIX Users' Group Conference*, Helsinki, Finland, May 1987.

[9] B. Martin, "The separation of interface and implementation in C++," in *Proceedings of the 3rd USENIX C++ Conference*, pp. 51–63, Washington, DC, USA, April 1991.

[10] E. Tuyen, W. Joosen, B. N. Jorgensen, and P. Verbaeten, "A generalization and solution to the common ancestor dilemma problem in delegation-based object systems," in *Dynamic Aspects Workshop (DAW '04)*, pp. 103–119, March 2004.

[11] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to express C++ concepts in Fortran 90," *ACM Fortran Forum*, vol. 15, pp. 13–18, 1997.

[12] V. K. Decyk, C. D. Norton, and B. K. Szymanski, "How to support inheritance and run-time polymorphism in Fortran 90," *Computer Physics Communications*, vol. 115, no. 1, pp. 9–17, 1998.

[13] E. Akin, *Object-Oriented Programming via Fortran 90/95*, Cambridge University Press, Cambridge, UK, 2003.

[14] D. W. I. Rouson, K. Morris, and X. Xu, "Dynamic memory deallocation in Fortran95/2003 derived type calculus," *Scientific Programming*, vol. 13, no. 3, pp. 189–203, 2005.

[15] M. Ljungberg, K. Otto, and M. Thuné, "Design and usability of a PDE solver framework for curvilinear coordinates," *Advances in Engineering Software*, vol. 37, no. 12, pp. 814–825, 2006.

[16] V. K. Decyk and H. J. Gardner, "Object-oriented design patterns in Fortran 90/95: mazev1, mazev2 and mazev3," *Computer Physics Communications*, vol. 178, no. 8, pp. 611–620, 2008.

[17] D. W. I. Rouson, J. Xia, and X. Xu, *Scientific Software Design: The Tao of SOOP*, Cambridge University Press, New York, NY, USA, 2010.

[18] K. Morris, D. W. I. Rouson, and J. Xia, "On the object-oriented design of reference-counted shadow objects," in *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE '11)*, pp. 19–27, May 2011.

[19] D. W. I. Rouson, K. Morris, and J. Xia, "Managing C++ objects with Fortran in the driver's seat: this is not your parents' Fortran," *Computing in Science and Engineering*, vol. 14, no. 2, pp. 46–54, 2012.

[20] D. W. I. Rouson, J. Xia, and X. Xu, "Object construction and destruction design patterns in Fortran 2003," in *Proceedings of the 10th International Conference on Computational Science (ICCS '10)*, pp. 1495–1504, June 2010.

[21] D. W. I. Rouson, J. Xia, and X. Xu, *Scientific Software Design: The Object-Oriented Way*, Cambridge University Press, New York, NY, USA, 2011.