IN-61

3 [illegible] 317

NASA Contractor Report 198431

An Object-Oriented Approach to Writing Computational Electromagnetics Codes

Martin Zimmerman and Paul G. Mallasch
Analex Corporation
Brook Park, Ohio

March 1996

Prepared for
Lewis Research Center
Under Contract NAS3-25776

National Aeronautics and
Space Administration

An Object-Oriented Approach to Writing Computational Electromagnetics Codes[*]

Martin Zimmerman and Paul G. Mallasch
Analex Corporation
Brook Park, Ohio 44142

Abstract

Presently, most computer software development in the Computational Electromagnetics (CEM) community employs the structured programming paradigm, particularly using the Fortran language. Other segments of the software community began switching to an Object-Oriented Programming (OOP) paradigm in recent years to help ease design and development of highly complex codes. This paper examines design of a time-domain numerical analysis CEM code using the OOP paradigm, comparing OOP code and structured programming code in terms of software maintenance, portability, flexibility, and speed.

Introduction

A great deal of work done in the Electromagnetics community in recent years involves development of numerical algorithms to solve EM problems quickly and accurately. The vast majority of existing CEM software written uses a "conventional" design; the Fortran programming language and a structured programming paradigm. Structured analysis and design concentrates more on processing with much less emphasis on the data being processed. But a number of CEM aspects make the task of writing numerical analysis quite challenging. These include:

- Parallelism;
- Efficient memory usage;
- Boundary conditions;
- Multiple coordinate systems;
- Hybrid methods using more than one numeric algorithm; and
- Handling rectangular structured, conformal structured, and unstructured grids.

Object-Oriented Programming (OOP) is a paradigm used to replace structured programming for the production of very large, complex computer programs. Object-oriented design contrasts conventional structured design by formally connecting code and data in a single entity known as an object. Object-oriented programming languages like C++ are rapidly gaining acceptance in many software communities. In some areas of numerical analysis, such as Computational Fluid Dynamics, object-oriented codes have made an appearance [1].

[*] This work was carried out at NASA Lewis Research Center under contract NAS3-25776.

How Do We Define Efficiency?

A goal of many programmers is efficient code. In the traditional programming paradigm efficiency meant speed of execution, but a better metric might be life-cycle costs of code. Besides execution time, life-cycle costs include time spent developing application code, time spent maintaining it, and time spent setting up any data files. Factoring money in, developers must know whether the end user is willing to pay for additional speed or a "friendlier" interface, for example. Numerous examples exist of situations where a user may take days (or longer) to build a model solved in hours on a modern supercomputer. Using the old metric, spending two man-months modifying code to run 50% faster made sense. Using a life-cycle metric this effort might not be cost effective. In addition, a life-cycle costs metric infers that slower-executing code that is easier to maintain may be more efficient than faster code that is difficult to maintain. This metric caused a move away from assembly languages towards high-level languages for application programs. In recent years, this same metric caused many software engineers to move from a structured programming paradigm to an OOP paradigm.

What Is OOP?

OOP is a programming paradigm; a conceptual model of computer software design using models organized around real-world concepts. The structured programming paradigm models a microprocessor. A series of "machines" called procedures or subroutines execute operations on data. In structured programming data and functions are separate entities; passive data (operated upon) and active functions.

In the real world the separation between data and function is not always clear. For example, an engineer is a person with a name and address, but also with skills that can be applied to a problem. An engineer is really a "thing" with both attributes (data) and methods (functions). In an OOP paradigm, these "things" are objects. Each discrete object has attributes and methods allowing it to accurately represent a specific concept incorporating both data structure and behavior. In OOP, data takes an active role because it knows what to do through its methods [2].

The OOP paradigm can be used to some extent in any language, even structured ones like Fortran [3]. However, hybrid OOP languages like C++ and pure OOP languages such as Smalltalk contain language constructs to allow binding together object attributes and methods easily.

Encapsulation

There are three main concepts in OOP: encapsulation, inheritance, and polymorphism. Encapsulation (also information hiding) involves packaging attributes and methods of an object together. Encapsulation wraps data in a layer of protective code, permitting access to data in an object only by the methods in that object. An object type (in C++ a "class") defines specific attributes and methods that an object possesses. When enforcing encapsulation, the outside world must access values of an object's attributes through its methods. Pure OOP languages require complete encapsulation while hybrid OOP languages let the programmer decide the level of encapsulation. Encapsulation hides implementation details of a high-level concept represented by an object from the outside world. The object becomes a "black box," limiting the possibility of errors like overwriting important data or applying functions incorrectly. A result is more maintainable and portable application code.

Object methods or services separate into several types. Use a constructor to create an object, similar to declaring a variable. A constructor allocates memory required for an object while creating the object, which can be at any point in the code. A destructor deletes an object, freeing up allocated memory and is normally called automatically when objects go out of scope. For example, an object created in a loop goes out of scope upon exiting the loop where a destructor deletes it. This dynamic memory allocation and deallocation makes memory management easier in C++ than it is in Fortran. Another service is the accessor, used by foreign objects to access data values inside an object. Accessors allow an object creator to give specific read or write permission to each object attribute. Many objects also contain utility services like a print method, for example. Instead of calling an external function that outputs an object's attributes, an object invokes a service causing the object to print itself.

Inheritance

OOP languages allow sharing of attributes and services among classes based on a hierarchical relationship. C++ uses the terms "base class" and "subclass." Inheritance permits a "child" subclass to receive all of its parent's attributes and methods. Usually the subclass represents a specialization of the base or super class. The definition of a subclass needs only to include those methods and attributes that are not part of the more generalized base class. In addition, designers may redefine subclass methods that are different from the super class. Services that do not change in the subclass definition are already automatically available.

Inheritance is a powerful property for code reuse. Carefully designed objects may be used over again in new applications, either as is, or as a base class for a new specialized object. Inheritance also allows hierarchical distribution of a complex concept across a number of classes. A good example in the present code is the Feed Class, which represents the forcing function applied to the EM time-domain boundary-value problem. The base Feed Class contains information common to all forcing functions. The primary service is application of a time-dependent source term to EM fields in a problem space. The time-dependent source term (e.g., pulse or sinusoid) is an attribute, as is the definition of a forcing function over a feed region. There are a number of specializations of the base Feed Class. For example, weighting across the feed region is different for a coax feed than for a rectangular waveguide. A plane-wave forcing function used in scattered field formulation has very different implementation from fixed region feed used in total field formulation. Subclass definitions contain these differences.

Polymorphism

Polymorphism means that the same named service may behave differently in different contexts. Users work at a high-level, using object methods to accomplish tasks without worrying about implementation details. If different forms of a single method accomplish the same task, then polymorphic functionality is present. The key to polymorphism is allowing run-time binding. Objects can invoke methods of other objects without knowing the type (or class) of the other object. The C++ language uses virtual functions for this purpose. While a strongly typed language, C++ uses base class pointers to point to a sub-class object. When invoking a virtual function through a base class pointer, the base object does not call its method, using instead the object method the pointer addresses. Polymorphism builds upon the previous two concepts. Encapsulation ensures that the outside world operates using an object method rather than working directly upon the object's attributes. Inheritance allows subclasses containing the same methods to use different implementations through polymorphism.

An example of polymorphism used in the present computer code is the Parallel Class. Parallel Class handles the work of spawning child processes, deciding which processes are neighbors, determining domain decomposition, and handling interprocess communication. Implementation is a function of the message passing library used (e.g., PVM or APPL), so a base Parallel Class can have several sub-classes. Each subclass corresponding to a different message passing library using library specific commands. For instance, a serial machine uses a default Parallel Class whose methods do nothing.

Algorithmic Polymorphism

Hybrid method computer codes are by definition polymorphic. Each implementation is capable of reaching the same result (accurate computation of EM fields) using different algorithms. For the present work, the hybrid method utilizes Finite Difference Time Domain (FDTD) and Finite Volume Time Domain (FVTD). Both methods use fields discretized over cell grids. Both methods (in at least some formulations) use similar processes (e.g., update E fields, step in time, update H fields, step in time, apply boundary conditions, save output, repeat). FDTD is much more memory efficient since it is not necessary to describe each cell individually. FVTD has the advantage of being applicable to unstructured grids as well as structured grids. Each algorithm is represented by a subclass of the same base class that contains the common methods (e.g., update E fields, etc.). An application of a hybrid FDTD or FVTD method is a complex conformal object surrounded by a regular grid containing the outer boundary radiating condition [4,5]. Using this hybrid method also could solve a problem entirely with FDTD or FVTD. By standardizing interfaces between regions using different algorithms, it is possible to have communication between these regions.

One implementation method divides the problem space into subspaces; each represented by an appropriate subclass object using either FDTD or FVTD. The subspaces collect into an array or list that contains base class pointers to subspace objects. Application code then traverses the list, invoking methods for each subspace object in turn without knowing how the object accomplishes its task.

Data Parallelism

Both FDTD and FVTD lend themselves to data parallelism implemented as domain decomposition. Hiding mechanics of domain decomposition allows each process operating on a subset of problem space to behave as if it is working on an entire problem. Application programmers and users also need to work on the entire problem combined. In the present case, Field Class and Focus Class hierarchies work together. Field Class contains discrete EM field data for the problem space. The data set is amorphous, neither structured nor unstructured. Focus Class provides framework for describing the ordering of Field Class data. An analogy is that Field Class is tent fabric and Focus Class is a set of tent poles defining the shape of the tent. Focus Object knows what subset of a Field Object to iterate over and how to iterate over that subset. This subset also is a function of data decomposition among processes. Focus Object takes position information listed globally and converts it to process local information used to control iterations over Field Classes (Figure 1). Since this algorithm implements structured and unstructured grids in different ways, these grids need varying sub-classes of Focus Class.

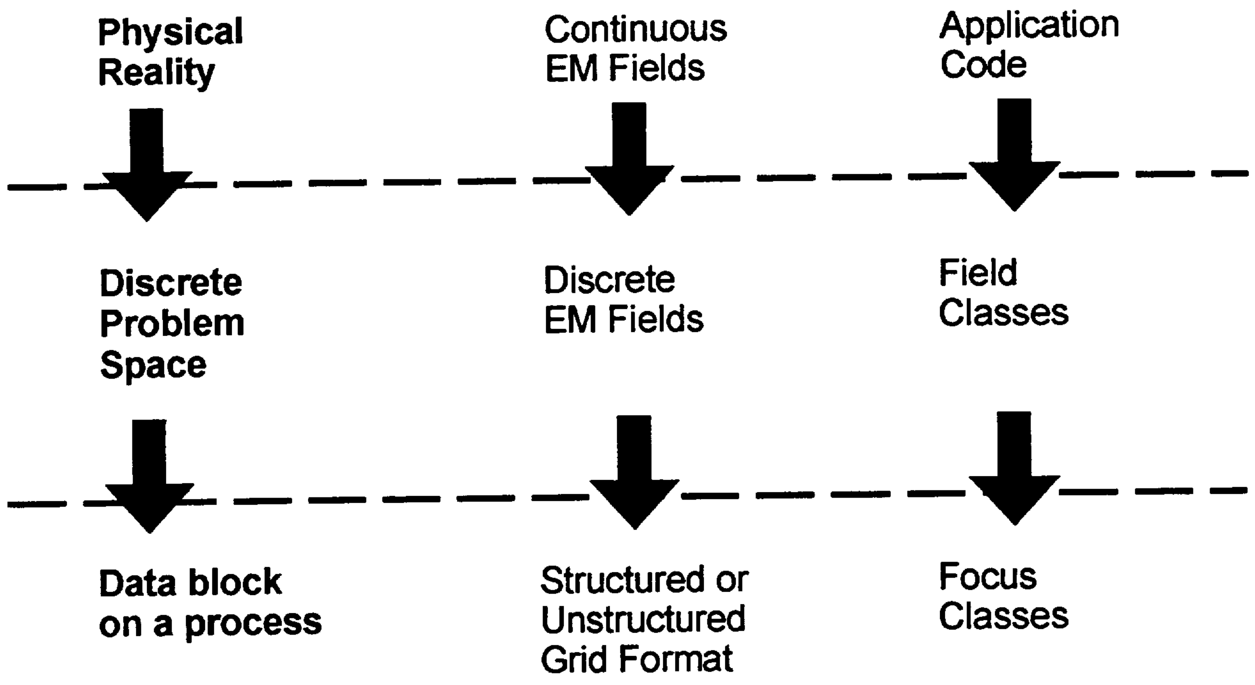

Figure 1 - Relationship Between Levels of Abstraction and the OOP Classes.

Both FDTD and FVTD use nearest neighbor information and therefore require message passing of EM field information between processes. Focus Class works in conjunction with Parallel Class for this task and Focus Object contains a Parallel Object as an attribute. Parallel Object determines whether data must be passed. If so, Focus Object loads a message with appropriate information and sends it using member functions (methods) of its Parallel Object.

C++ dynamic memory allocation is very useful for data parallelism. Dimensioning arrays to the correct size as a function of the number of processes used. The software uses a minimum amount of memory without recompilation after adjusting the number of processors. This is particularly important for people using clusters of workstations that have load distributions that vary from day to day.

Boundary Conditions

Boundary conditions often represent one of the most difficult aspects of numerical analysis programming. They frequently demand a small fraction of execution time while comprising a large part of actual source code. The present code represents boundary condition concepts by a class hierarchy. All boundary conditions provide the same service; modification of fields in a subset of problem space. In addition, they all use data that are primarily multiplicative constants required by the boundary condition. Actual data and implementation vary in different boundary conditions. Handle differences in implementation by creating distinct sub-classes representing miscellaneous boundary conditions, such as first order Mur or second order Liao.

One challenge of using boundary conditions is applying operations to only a portion of EM fields in the problem space. Focus Object handles iteration for the data set in Field Object and tracks the active region. This service also applies forcing functions and material attributes (Figure 2). The boundary condition may contain smaller Field Objects that cover only the boundary region. Focus Object insures that mathematical operations on two Field Objects only occur if objects have the same size active region.

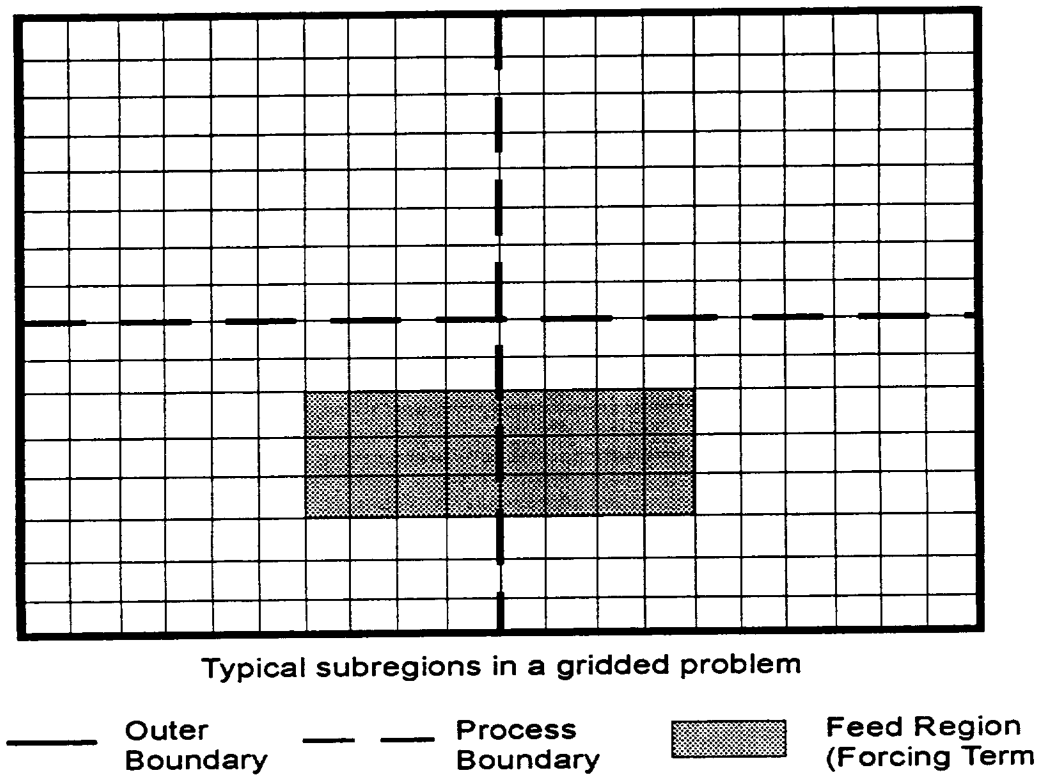
Typical subregions in a gridded problem
Outer Boundary
Process Boundary
Feed Region (Forcing Term

Figure 2 - Various subsets of the entire grid must be operated on during numerical analysis.

Comparisons With A Structured Programming Code

Comparing the present C++ code to FDTD_ANT (a non-conformal structured-grid FDTD program based upon the Luebbers and Kunz FDTDA code [6]), both codes use the same FDTD formulation executing with either scattered field or total field formulation. These codes are compared for development, maintenance, flexibility, portability, memory efficiency, and speed efficiency. A version of FDTD_ANT is parallelized for domain decomposition along any one of three Cartesian axes. The C++ code may be domain decomposed along any combination of three axes (including all three at once).

Code Development

Writing object-oriented code often involves more design overhead than writing a similar code in the structured programming paradigm. In Object-Oriented Design (OOD) the problem domain must be divided into discrete, distinguishable entities modeled by objects. The software engineer designs objects or a hierarchy of objects and relationship between objects. In some cases, classes may already exist containing some or all the necessary capabilities. Using existing classes as a foundation for new classes is a hallmark of reusable software--reducing design, coding, and testing costs by amortizing development over several designs. Code reuse can simplify development by using classes already created and tested in different applications. Code reuse is different from duplicating code or taking portions of code from one file and copying it into another file. Reuse within an application requires finding similarities and consolidating them using inheritance. Practicing software reuse can mean developing and debugging less total code for a new application with fewer opportunities for typographical and logic errors.

Code Maintenance

Code maintenance is a major issue in life-cycle costs since most CEM codes are undergoing almost constant modification. The OOP paradigm specifically addresses code maintenance in several ways. Encapsulation makes code more modular, which helps prevent new code from affecting existing and unrelated code. In addition, much less global data (including common blocks) exists in the present software, making overwritten memory less likely. Additional code is usually simpler to write. For example, iterator indices of Field Object are all encapsulated within Focus Object and encapsulated within Field Object is Focus Object. Application code using Field Objects contain no indices or "do-loops." Index-free notation at the application code level makes code easier to read and avoids potential for typographical errors that often accompany nested "do-loops."

Flexibility

The OOP code is far more flexible due to polymorphism. Using class hierarchies containing virtual functions and functors (functions that are really objects and treated like data) simplifies writing hybrid method codes. In particular, hybrid method codes do not appear to use multiple methods or algorithms. This technique increased software versatility by adding new subclasses that define only methods whose implementation has changed.

Portability

One clear portability advantage that emerged involved parallelization. The Fortran code has both a serial version and a parallel version. Executing a parallel version on a serial machine necessitates removing all message passing library calls since that library is not installed. In the OOP paradigm, each computer type would have a library containing the Parallel subclass relevant to the computer's configuration. For instance, a serial machine would have a "stub" subclass that would do nothing. Application code dealing only with the base Parallel Class becomes completely portable. This portability applies easily to other classes, with highly optimized algorithm kernels stored in class libraries that vary from machine to machine. Application codes invoke methods in these libraries without knowing implementation details.

Memory Efficiency

Fortran tends to be very memory inefficient due toU the lack of dynamic memory allocation. This is a problem for parallel codes where the data set size is a function of the number of processes running. The C language has dynamic memory allocation through the malloc command, but an additional command must deallocate memory. C++ goes a step further by automatically invoking destructors aiding memory deallocation.

Speed Efficiency

While the project has not concentrated on execution speed, efficiency remains an important issue. Originally, OOP code ran significantly slower than structured Fortran code--a factor of 8.5:1. In reference [7], the authors compared versions of a Computational Fluid Dynamics (CFD) code written in Fortran, C, and C++. The first two codes used structured programming while the C++ code used OOP. On a variety of computing platforms C code was faster than C++ code by a factor of roughly 2. Fortran code was nearly equivalent to C code in speed. For many numerical applications, a factor of 2:1 would be an acceptable loss in execution efficiency. The work in [7] suggests that the present OOP code could execute faster. Following modification, later runs produced an overall C++ to Fortran ratio of 1.7:1. This includes a different (and slower) Boundary Condition used by the C++ software. In main interior field update routines, the ratio is 1.3:1 (on an IBM RS6000/580).

Results

To date, OOP code has been written in C++, fully parallelized (3-dimensional domain decomposition) using the PVM message passing library. The software uses the FDTD algorithm on a structured grid with either PML or first order Mur boundary conditions. Agreement with the FDTD_ANT code is good for sample problems, but is not precise since the codes use different boundary conditions. At present the unstructured grid classes remain a complicated source code problem. Adding an FVTD solver resulted in complete algorithmic polymorphism, but the ensuing code executed extremely slowly and requires further optimization.

References

[1] I. G. Angus and W. T. Thompkins, "Data storage, concurrency and portability: an object oriented approach to fluid mechanics," The Fourth Conference on Hypercubes, Concurrent Computers, and Applications, 1989.

[2] J. Dutemann, "Our object all sublime," PC Techniques, pp. 14-22, April/May 1990.

[3] K. D. Wampler, "The object-oriented programming paradigm (OOPP) and Fortran programs," Computers in Physics, pp. 385-394, July/August 1990.

[4] K. S. Yee, J. S. Chen, and A. H. Chang, "Conformal finite difference time domain (FDTD) with overlapping grids," IEEE Trans. Antennas Propagat., vol. 40, no. 9, pp. 1068-1075, Sept. 1992.

[5] K. S. Yee and J. S. Chen, "Conformal hybrid finite difference time domain and finite volume time domain," IEEE Trans. Antennas Propagat., vol. 42, no. 10, pp. 1450-1455, Oct. 1994.

[6] J. H. Beggs, R. J. Luebbers, K. S. Kunz, and H. S. Langdon, "User's Manual for three dimensional FDTD version A code for scattering from frequency-independent dielectric materials," short course at the 8th Annual Review of Progress in Applied Computational Electromagnetics, March 1992.

[7] I. G. Angus and J. L. Stolzy, "Experiences in converting an application from Fortran to C++: Beyond f2c," Proceedings of the C++ at Work Conference, Nov. 1991.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE March 1996	3. REPORT TYPE AND DATES COVERED Final Contractor Report

4. TITLE AND SUBTITLE An Object-Oriented Approach to Writing Computational Electromagnetics Codes	5. FUNDING NUMBERS WU-233-5A-5C C-NAS3-25776
6. AUTHOR(S) Martin Zimmerman and Paul G. Mallasch	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Analex Corporation 2001 Aerospace Parkway Brook Park, Ohio 44142	8. PERFORMING ORGANIZATION REPORT NUMBER E-10172
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA CR-198431

11. SUPPLEMENTARY NOTES

Project manager, Charles Raquet, Space Electronics Division, NASA Lewis Research Center, organization code 5640, (216) 433-3471.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 61 This publication is available from the NASA Center for AeroSpace Information, (301) 621-0390.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Presently, most computer software development in the Computational Electromagnetics (CEM) community employs the structured programming paradigm, particularly using the Fortran language. Other segments of the software community began switching to an Object-Oriented Programming (OOP) paradigm in recent years to help ease design and development of highly complex codes. This paper examines design of a time-domain numerical analysis CEM code using the OOP paradigm, comparing OOP code and structured programming code in terms of software maintenance, portability, flexibility, and speed.

14. SUBJECT TERMS Object oriented programming; Computational electromagnetic software simulation	15. NUMBER OF PAGES 12
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT