

# Object-Based Programming in Fortran 90

Mark G. Gray<sup>1</sup>      Randy M. Roberts<sup>2</sup>

April 15, 1997

<sup>1</sup>Mark G. Gray (gray@lanl.gov) is a member of the Radiation Transport Team in the Scientific Computing Group at Los Alamos National Laboratory

<sup>2</sup>Randy M. Roberts (rsqrd@lanl.gov) is a member of the Radiation Transport Team in the Scientific Computing Group at Los Alamos National Laboratory

## 1 Introduction

We are object-oriented enthusiasts. We delight in analyzing our computer modeling problems with the tools of composition and classification that scholars have used since the time of Aristotle. We relish designing solutions to these problems using the concepts and entities of the problem domain instead of being restricted to the concepts and entities of the computer language. Finally, we enjoy implementing our designs in languages that fully support our object-oriented analysis and design.

Our object-oriented language of choice is C++. We find its unity of design, reasonable implementation of features, and concerns for efficiency perfect for scientific programming. All of this is as its creator intended: Stroustrup clearly identifies these features as his design goals[1] and we believe he achieved them.

Unfortunately, we aren't always afforded the luxury of using an object-oriented language. Availability on a given platform, interfacing requirements, and performance prejudices sometimes restrict the languages we are allowed to use.

Fortunately, object-oriented analysis and design can be fruitfully applied to problems even when their ultimate implementation is in a non-object-oriented language. In fact, if a language supports user-defined types, then an object-oriented design can be implemented in it in an object-based fashion.

We have developed strategies for doing object-based programming in Fortran 90. We are certainly not the first to do this. Meyer[2] and Rumbaugh[3] both devote a chapter in their books to object-based programming in C, and Fortran 77; the latter provides the outline we will follow. Norton[4] provides a web site<sup>1</sup> of interesting object-based Fortran 90 examples. We offer our techniques here in the hopes of helping fellow object-oriented enthusiasts cope with implementation in a popular non-object-oriented language, and in the hopes of clarifying just what object-oriented programming is for those who may know Fortran 90 and wonder what the object-oriented hoopla is all about<sup>2</sup>.

---

<sup>1</sup><http://www.cs.rpi.edu/~nortonc/oof90.html>

<sup>2</sup>Throughout this paper we will use several implicitly defined Fortran 90 and object-oriented terms. For explicit definitions of these terms consult Brainard[5] and Booch[6].

## 2 Object-Oriented and Object-Based Programming

Rumbaugh defines object-oriented programming as programming in terms of a collection of discrete objects that incorporate both data and behavior[3]. In order to be object-oriented a language *must* support these four features:

- Identity — the quantization of data in discrete, distinguishable entities called objects
- Classification — the grouping of objects with the same structure and behavior into classes
- Polymorphism — the differentiation of behavior of the same operation on different classes
- Inheritance — the sharing of structure and behavior among classes in a hierarchical relationship

Modern languages that provide user-defined types can provide identity and classification, and some even support polymorphism. However without inheritance these languages are not object-oriented. Cardelli and Wegner identify using user-defined types for identity and classification without inheritance as object-based programming[7]. Since Fortran 90 lacks inheritance it is not an object-oriented language; however, its user-defined types permit its use as an object-based language.

In the next section we show the analysis and design of a `stopwatch` class. In Section 4 we illustrate the steps necessary to implement the `stopwatch` class in an object-based fashion in Fortran 90. In Section 5 we revisit the analysis and design and look for code improvements. Finally we comment on this approach.

## 3 Analysis and Design

We wish to develop a code timing utility. It should record the total execution time and time spent in various sections of code. The physical object we want to model is the modern electronic stopwatch, which records total and split times.

We'd like the stopwatch to be used like this:

```
call split(s, "bar") ! turn "bar" split on
call bar()           ! execute bar subroutine
call split(s, "bar") ! turn "bar" split off
```

```

call split(s, "foo") ! turn "foo" split on
call foo()           ! execute foo subroutine
call split(s, "foo") ! turn "foo" split off
call report(s, 6)    ! report total and split times

```

and produce output like this:

```

                                TIMER STATISTICS (sec)
                                0----1----2----3----4----5----6----7----8----9----%
bar   0.152 :*****
foo   0.252 :*****
                                0----1----2----3----4----5----6----7----8----9----%
total 0.404

```

First, we analyze this problem statement for the key abstractions that will become the classes in our implementation. One obvious class is the `stopwatch` itself, which first maps names to elapsed times and then reports its associated data. Less obvious, but perhaps more fundamental, is the basic `timer`, which records the elapsed time. The `timer` class does this by consulting the system clock at the start and end of a timed interval. The total and split times are recorded by individual `timers`.

Next, we design the associations and behaviors of these classes. The `stopwatch` class is composed of several `timers`, each with an associated name, so the `stopwatch` class must keep a set of character strings and corresponding `timers`. The `split` method of the `stopwatch` class toggles the state of the named `timer`. Finally the `report` method of the `stopwatch` class reports total and split times by name. The `timer` class must keep track of the elapsed time while it is on; its users need some means to turn it on and off. We call this method `switch`; if the `timer` is off `switch` will turn it on and vice versa. A `timer` must also report its elapsed time via a method called `report`.

Finally, we implement our design in Fortran 90 in the following section.

## 4 Implementation

To use Fortran 90 as an object-based language we must map its features into object-oriented concepts. Rumbaugh[3] identifies the following steps necessary to object-based programming:

1. Translate classes into structures

2. Pass arguments to methods
3. Allocate storage for objects
4. Implement method resolution
5. Implement associations
6. Encapsulate internal details of classes
7. Implement inheritance in data structures

In this section we show how this can be done in Fortran 90, using the `stopwatch` and `timer` classes as examples.

#### 4.1 Translate classes into structures

Fortran 90 introduces structures under the rubric of derived types. Since “derived” has a very different meaning in object-oriented parlance, we will use the more generic term “user-defined”. Classes can be based on user-defined types, with class attributes corresponding to the components of the user-defined type. Creating a variable of the user-defined type corresponds to creating an object of a particular class.

The `stopwatch` class is based on the `stopwatch` user-defined type:

```
integer, parameter, private :: mnl = 10

type, public :: stopwatch
  private
  integer :: max_splits
  integer :: free
  character(len = mnl), pointer, dimension(:) :: name
  type(timer), pointer, dimension(:) :: split
end type stopwatch
```

where the `timer` class is based on the `timer` user-defined type:

```
type, public :: timer
  private
  integer :: time
  logical :: on
end type timer
```

Each `timer` has an `integer::time`<sup>3</sup> that records its elapsed time, and a `logical::on` flag which indicates whether it is on. A `stopwatch` consists of an array of character strings, called `name`, and `timers`, called `split`, with integers `max_splits` and `free` for the maximum and actual size of the arrays, respectively.

By placing the attributes of a `stopwatch` or `timer` together in a user-defined type we have achieved the following desirable programming objectives:

**Abstraction** — users needing timing information deal with one `stopwatch` variable instead of arrays of times and names. If the user needs several stopwatches, he simply creates several distinct variables of type `stopwatch`.

**Type safety** — compilers can check that `stopwatch` variables are used *as* stopwatches in procedures.

**Encapsulation** — implementors can easily create or change procedures that use stopwatches since their components are grouped together.

Variables of the user-defined `stopwatch` and `timer` type have state and identity, but lack any intrinsic behavior necessary for object-hood. We add behavior by adding methods to obtain genuine objects.

## 4.2 Pass arguments to methods

Our Fortran 90 user-defined types are given behavior by adding methods: subroutines and functions which operate on a variable of the user-defined type passed as the first parameter<sup>4</sup>. We use the naming conventions of prepending the class name and underscore to the method name to further associate it with the class, and of naming the corresponding user-defined type parameter `self`, in keeping with the convention of some well known object-oriented languages<sup>5</sup>.

The `stopwatch` user-defined type becomes a `stopwatch` class by adding the following methods:

```
subroutine stopwatch_construct(self)
```

---

<sup>3</sup>computed from the Fortran 90 intrinsic `SYSTEM_CLOCK`, which returns integers `COUNT`, `COUNT_RATE`, and `COUNT_MAX` based on the system clock

<sup>4</sup>Fortran 90 passes arguments by reference; large user-defined types can be used as parameters without concern over copy overhead.

<sup>5</sup>C++ uses `this` instead of `self`.

```
subroutine stopwatch_construct_1(self, n)
subroutine stopwatch_split(self, name)
function stopwatch_resolution()
subroutine stopwatch_report(self, u)
subroutine stopwatch_destruct(self)
```

The `stopwatch_construct` and `stopwatch_construct_1` subroutines prepare a stopwatch variable by allocating its name and timer arrays and initializing that data. The `stopwatch_destruct` subroutine deallocates these arrays. Once defined and initialized, the `stopwatch_split` routine toggles a named `split`. The `stopwatch_resolution` function is somewhat different from the others; it is a class method instead of an object method. A class method affects the behavior of the entire class; it takes no `self` parameter and does not rely on the existence of a class variable to perform its job. The `stopwatch_resolution` returns the resolution of the `stopwatch` class, which is data common to all `stopwatch` objects. The `stopwatch_report` subroutine reports the times recorded by the `split` array to logical unit `u`.

The timer user-defined type becomes a timer class by adding the following methods:

```
subroutine timer_construct(self)
subroutine timer_switch(self)
function timer_is_on(self)
function timer_time(self)
function timer_resolution()
subroutine timer_destruct(self)
```

The `timer_construct` subroutine prepares a timer variable by initializing its data. The `timer_destruct` subroutine is not really needed here since no deallocation is necessary, but is included in case a change in the implementation of `timer` required it. Once defined and initialized, the `timer_switch` routine toggles a timer. The `timer_is_on` function returns the timer's state, and the `timer_time` function returns a timer's elapsed time. The `timer_resolution` function is another class method which returns the resolution of a timer.

The behavior given to the `stopwatch` and `timer` user-defined types by these methods make them true classes; variables of their types are true objects.

By adding methods to the user-defined types, we have achieved the following desirable programming objectives:

**Encapsulation** — users of the `timer` class need know nothing about its user-defined type's components, nor of its procedure's implementation in order to use it. Similarly the developer of the `timer` class is free to change its internal attributes and method coding as long as he maintains the promised behavior of the class.

The `timer` class underwent several major changes during the course of its development; because of this encapsulation, its user (the `stopwatch` class) remained unchanged. The user is freed from having to worry about how the class works, and the programmer is freed from having to worry about how the users uses it.

### 4.3 Allocate storage for objects

Fortran 90 supports static allocation (global variables), automatic allocation (local variables), and heap allocation (using the `allocate` keyword). Unfortunately Fortran 90 does not automatically provide for the allocation (or deallocation) of user defined type components through the invocation of user-defined constructors (or the destructor). We compensate for this by giving each class at least one `construct` method, which is responsible for any allocation and initialization, and one `destruct` method, which is responsible for any deallocation. The user is responsible for calling one of the constructor subroutines to properly initialize the class, and calling the destructor routine for deallocation after use. This detail is already taken care of in most object-oriented languages, where construction is an automatic part of variable declaration and destruction is an automatic part of variables leaving scope.

The `stopwatch` “default” constructor, `stopwatch_construct`, merely allocates space for 20 `split` and 20 `character` strings in `name` via a call to the `stopwatch_construct_1` constructor<sup>6</sup>:

```
subroutine stopwatch_construct(self)
  implicit none
  type(stopwatch), intent(inout) :: self

  call stopwatch_construct_1(self, 20)

end subroutine stopwatch_construct
```

---

<sup>6</sup>We use the `intent` keyword to restrict the use of arguments to procedures. The `intent(in)` argument attribute prohibits a procedure from modifying the argument's content. This corresponds to the `const` keyword in C++.

The `stopwatch_construct_1` subroutine allocates the specified storage in `name` and `split` and then calls each timer's `construct` subroutine to let the timers initialize themselves:

```

subroutine stopwatch_construct_1(self, n)
  implicit none
  type(stopwatch), intent(inout) :: self
  integer, intent(in) :: n

  integer :: i

  ! make n names, splits
  self%max_splits = n
  allocate(self%name(0:self%max_splits))
  allocate(self%split(0:self%max_splits))

  ! blank all names, zero all splits
  do i = 0, self%max_splits
    self%name(i) = "      "
    call construct(self%split(i))
  end do

  ! turn total timer on
  self%free = 0
  self%name(0) = "total"
  call switch(self%split(0))

end subroutine stopwatch_construct_1

```

The `stopwatch` destructor, `stopwatch_destruct`, calls each timer's `destruct` subroutine, and then deallocates the `name` and `timer` arrays:

```

subroutine stopwatch_destruct(self)
  implicit none
  type(stopwatch), intent(inout) :: self

  integer :: i

  ! destroy each split
  do i = 0, self%max_splits

```

```

        call destruct(self%split(i))
    end do

    ! deallocate split and name arrays
    deallocate(self%split)
    deallocate(self%name)
    self%max_splits = 0
    self%free = 0

end subroutine stopwatch_destruct

```

By adding constructors and the destructor to our class, we have achieved the following desirable programming objectives:

**Encapsulation** — users can ensure that `stopwatches` are constructed in a valid state. Additionally, users can readily track resources since resource (de)allocation is localized within the class destructor and constructors.

Initialization is an important part of every language, and object initialization by the constructor is especially important in an object approach.

#### 4.4 Implement method resolution

Since the first argument to a class method is the `self` argument, the compiler can resolve a class method by its argument signature. By placing these methods in interface blocks and giving them generic names we get automatic method resolution, a form of polymorphism, which makes it easier for the user to know what methods are available. For example, a `stopwatch` object can be constructed with a default number of `splits` via `stopwatch_construct`, or with a specified number of `splits` via `stopwatch_construct_1`. If we interface these constructors to the generic name `construct` all the user need remember is **how** to construct a `stopwatch` (what parameters are needed), not **what** the method is called.

For the `stopwatch` class these interfaces look like this:

```

interface construct
    module procedure stopwatch_construct, stopwatch_construct_1
end interface

interface split

```

```

        module procedure stopwatch_split
    end interface

    interface report
        module procedure stopwatch_report
    end interface

    interface destruct
        module procedure stopwatch_destruct
    end interface

```

Following this convention lets users of any object build, use, and destroy it in a consistent fashion. For example, a `stopwatch` object named `s` is declared and initialized as follows:

```

    use stopwatch_class ! get stopwatch type definition and methods
    type(stopwatch) :: s ! declare a stopwatch variable
    call construct(s)   ! initialize it

```

where the call to the generic routine `construct` resolves to a call to the `stopwatch_construct` routine because of its argument signature. The `stopwatch` object `s` is uninitialized as follows:

```

    call destruct(s) ! clean up s

```

By adding generic constructors and a generic destructor to our class, we have achieved the following desirable programming objectives:

**Polymorphism** — users can construct and destruct `stopwatches` and `timers` in a consistent manner, i.e. by calling the `construct` and `destruct` generic methods. Further, developers can use consistent method names without concern about name conflicts between classes.

## 4.5 Implement associations

We implement associations via membership of types or pointers to types. The `timer` class is **part-of** a `stopwatch`, and this composition is expressed by the array of `timers` contained in the `stopwatch` user-defined type. In Subsection 4.7 we will create a `serial_stopwatch` class that is a **kind-of** `stopwatch`, and this classification is expressed by pointer containment.

By implementing associations, we have achieved the following desirable programming objectives:

**Composition** — developers can build new classes using existing classes as parts.

**Classification** — developers can build new classes by specializing from existing classes.

#### 4.6 Encapsulate internal details of classes

Although we have tied the state of a `stopwatch` to its behavior by requiring that all `stopwatch` methods have a `stopwatch` as their first type, we can further enforce this relationship by using Fortran 90 modules. The Fortran 90 module can contain user-defined types, variables, and procedures, with user specified access.

The `stopwatch` class is encapsulated in the `stopwatch_class` module:

```
module stopwatch_class

    use timer_class
    public :: construct, split, stopwatch_resolution, report, destruct
    private :: stopwatch_construct, stopwatch_construct_1, &
        stopwatch_destruct, stopwatch_split, stopwatch_report

    ! stopwatch type declaration goes here...

    ! interface statements go here...

contains

    ! method definitions go here...

end module stopwatch_class
```

Here users of the `stopwatch_class` module can call the subroutines `construct`, `split`, `stopwatch_resolution`, `report`, and `destruct` because they are all declared `public`; users have no access to any of the routines declared `private`. Users can also create variables of type `stopwatch` because the type is `public`, but cannot access any of the member data of that type since it is all declared `private`.

By using Fortran 90 modules, we have achieved the following desirable programming objectives:

**Encapsulation** — developers can find all of the `stopwatch` class data and behavior in one location, the `stopwatch_class` module. Users of `stopwatch` class need only use the `stopwatch_class` module to obtain all of the data and behavior of the class.

**Data hiding** — users do not need access to the internal workings of the `stopwatch` class. Developers are free to use whatever attributes and implementation they want, as long as they adhere to the expected behavior of the class' public interface.

#### 4.7 Implement “inheritance” in data structures

Inheritance is arguably the most important concept in science. Knowing that an emu is a kind of bird tells me many things about its behavior, even if I don't know anything else about emus. Similarly, inheritance is arguably the most important concept in object-oriented programming. Knowing that a `symmetric_matrix` is a **kind-of** `matrix` tells me that I can expect of it all of the behavior of a `matrix`, and that I can use it wherever I would use a `matrix`, even if I don't know anything else about `symmetric_matrixes`. When the natural inheritance structure of the real world is built into an object-oriented model, the important relationships known about the real world carry over naturally into the model, providing developers and users powerful tools for managing the complexity of the model.

Unfortunately Fortran 90 does not support inheritance. Fortunately some of the features of inheritance can be faked in Fortran 90, giving some of its benefits.

As a simple example of the use of inheritance, we will extend the `stopwatch` class to support timing on a parallel platform. We want the interface of the parallel `stopwatch` to be identical to that of the serial `stopwatch` and prefer not to have to worry at all about which `stopwatch` we are using. Since a parallel `stopwatch` is a **kind-of** `stopwatch`, and a serial `stopwatch` is a **kind-of** `stopwatch`, what we want here is inheritance: The serial and parallel versions are merely variations on the main theme of `stopwatch`.

To implement inheritance in the `stopwatch` class we first globally replace “`stopwatch`” in the previous sections with “`serial_stopwatch`”. The previously defined `stopwatch` class becomes the `serial_stopwatch` class, which is a **kind-of** `stopwatch`. The `parallel_stopwatch` is a **kind-of** `stopwatch` too; it is similar to its serial sibling except for its `construct` and `report` methods. The `parallel_stopwatch`'s `construct` method establishes the communication pattern between the identical `parallel_stopwatch` objects

when they are constructed on each processor. The `parallel_stopwatch`'s `report` method uses the communication pattern established by `construct` to reduce the multiple `parallel_stopwatch` object's data (via scatter-with-operation, where the operation may be sum, average, maximum, minimum, or some combination of these) for output.

Finally, we write the generic stopwatch class. Its type declaration is as follows:

```
integer, parameter, private :: SERIAL = 0, &
                                PARALLEL = 1

type, public :: stopwatch
  private
  integer :: type
  type(serial_stopwatch), pointer :: s
  type(parallel_stopwatch), pointer :: p
end type stopwatch
```

The constructor for the stopwatch class allocates the appropriate type based on whether the platform is parallel or not:

```
subroutine stopwatch_construct(self)
  implicit none
  type(stopwatch), intent(inout) :: self

  self%type = platform()      ! returns SERIAL or PARALLEL
  select case(self%type)
  case (SERIAL)
    allocate(self%s)          ! allocate serial_stopwatch
    call construct(self%s) ! and initialize it
    nullify(self%p)
  case (PARALLEL)
    self%type = PARALLEL
    nullify(self%s)
    allocate(self%p)          ! allocate parallel_stopwatch
    call construct(self%p) ! and initialize it
  case default
    ! abort with error message
    ...
  end select

end subroutine stopwatch_construct
```

All the other `stopwatch` methods test the `type` component, and invoke the appropriate method for the allocated member. For example, the `stopwatch_report` method:

```
subroutine stopwatch_report(self)
  implicit none
  type(stopwatch), intent(in) :: self

  select case (self%type)
  case (SERIAL)
    call report(self%s)
  case (PARALLEL)
    call report(self%p)
  case default
    ! abort with an error message
    ...
  end select

end subroutine stopwatch_report
```

calls either `serial_stopwatch_report` or `parallel_stopwatch_report` depending on the **kind-of** `stopwatch` that the `stopwatch` object really is.

By using inheritance, we could have achieved the following desirable programming objectives:

**Classification** — in **true** object-oriented languages users can define procedures that take `stopwatch` objects as arguments without concern about which **kind-of** `stopwatch` is passed in. Objects of these child classes can be used *wherever the base class is expected!* In our Fortran 90 implementation one cannot use an instance of `serial_stopwatch` wherever a `stopwatch` is required.

**Polymorphism** — in **true** object-oriented languages users can call the `report` method on an instance of the `stopwatch` class and get the correct behavior, based on the actual type. We have also achieved this objective with our Fortran 90 implementation.

**Extendibility** — in **true** object-oriented languages one can add a new child class *without changing existing base class code*. In our Fortran 90 implementation developers of new classes derived from the `stopwatch`

class must modify the base `stopwatch` class to accommodate the new type.

**Reuse** — in **true** object-oriented languages the base class would contain the code common to most of the classes within the **kind-of** hierarchy. Particular child classes can override this common code. In our **Fortran 90** implementation we must duplicate identical code in each of the child classes in order to allow future classes to override the default behavior.

**Efficiency** — in **true** object-oriented languages the compiler can use efficient pointer indirection for child class polymorphism, and in some cases can optimize the indirections away. In our **Fortran 90** implementation the compiler must deal with the **case** statement.

## 5 Iteration

One of the major advantages of the separation of interface (what the class can do) and implementation (how it does it) in an object-oriented approach is the ease with which design iteration can be done. Design improvements that do not change the interface can be implemented without changing any of the client code. Several of these changes can be made to improve the `stopwatch` class.

First, the 10 character limit placed on names in the `stopwatch` class could easily be removed by simply making `name` an array of `character` pointers and allocating the exact length needed when a new name is seen. Along these same lines `name` and `split` are fixed length arrays allocated by `construct`; they could instead be implemented as linked lists which dynamically grow as needed.

Next, the association between the `name` and `split` arrays follows a well know pattern called a mapping or dictionary; in **C++** the `stopwatch` class could be instantiated from an STL `map` class[8]. We could base our `stopwatch` class on a similarly generic `map` class in **Fortran 90** (with the help of a preprocessor like `m4` to parameterize the `map`) and gain the possibility of additional code reuse.

Finally, the `parallel_stopwatch` class could be extended in a variety of interesting ways. It currently collates and reports the minimum, average, and maximum time for each (parallel) named split; it could instead present its output in more interesting ways. If beginning and end time arrays were added to the `timer` class the `parallel_stopwatch` class could

report a complete execution time line for each of the processors. The last change probably call for another child class in the `stopwatch` hierarchy.

## 6 Conclusion

Most of the advantages of object-oriented programming carry over to object-based programming in Fortran 90, at the expense of extra work and discipline. Are the benefits worth the expense?

We think so, for two reasons. First, even with the extra work, we find the object-based approach puts us ahead at the end of the day. Once its specification is complete, each object becomes an independent programming project; a large code naturally decomposes into smaller codes. Our ability to rapidly get small codes running permits us to tackle large codes more easily. *Divide et imperia* is the best way to manage complexity. The resulting implementations *are* cleaner, more robust, more extendible, and more maintainable. Second, Fortran 90 is a language with a quirky, powerful, non-orthogonal feature set that presents many pitfalls for the unsuspecting programmer. To learn and use all of its features well require much work and discipline regardless of the programming approach. Object-based programming provides one framework for mastering these new features.

In practice we use all of the above techniques, except inheritance. We find this pale imitation of `true` inheritance requires far too much work for far too little benefit. The ANSI Fortran 2000 committee has recognized the value of `true` inheritance, and has included it in its draft standard.

Given the choice between using Fortran 90 or an object-oriented language Rumbaugh[3] succinctly summarizes our views:

Use of an object-oriented or non-object-oriented language is not a matter of functionality. By using the mappings described above, you can translate any object-oriented construct into a non-object-oriented language. Computational power is never an issue because any universal language can compute anything computable.

The real issue with languages is not power but expressiveness, convenience, protection from errors, and maintainability. An object-oriented language makes writing, maintaining, and extending programs easier and safer because it performs tasks that the non-object-oriented language programmer must perform manually.

Rumbaugh then goes on to conclude:

Nevertheless, if you must use a non-object-oriented language, we feel that an object-oriented design will simplify your task and provide greater flexibility and extensibility if you are willing to program in a disciplined manner.

## References

- [1] Stroustrup B. *The Design and Evolution of C++*, Addison-Wesley, 1994
- [2] Meyer B. *Object-oriented Software Construction*, Prentice-Hall, 1988
- [3] Rumbaugh J., Blaha M., Premerlani W., Eddy F., and Lorensen W., *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
- [4] Norton C., Szymanski B., and Decyk V., *Object Oriented Parallel Computation for Plasma Simulation*, Comm. of ACM, 38(10) Oct 1995
- [5] Brainerd W. S., Goldberg C. H., Adams J. C., *Programmer's Guide to Fortran 90*, Springer Verlag, 1996
- [6] Booch G. *Object-Oriented Analysis and Design with Applications, 2nd ed.*, Benjamin/Cummings, 1994
- [7] Cardelli, L. and Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys, 17(4), 471–522 1985
- [8] Musser D. R., and Saini A., *STL Tutorial and Reference Guide*, Addison-Wesley, 1996