# Notes on the programming style in CHAMP

Julien Toulouse
Laboratoire de Chimie Théorique
Université Pierre et Marie Curie et CNRS, 75005 Paris, France
julien.toulouse@upmc.fr

July 2, 2016

These notes describe the programming paradigm used in the Fortran 90 files of the program CHAMP. This programming paradigm was invented by the theoretical chemist and legendary programmer François Colonna and was given different names over the years: Open Structured Interfaceable Programming Environment (OSIPE) [**?**], Deductive Object Programming [**?**], and Implicit Reference to Parameters (IRP) [**?**]. Even though we describe a Fortran implementation of it, this programming paradigm is not restricted to any particular programming language, and in fact has been implemented in many languages. The particular Fortran implementation that I made in CHAMP was very much inspired by the implementation in the program QMCMOL [**?**] mainly made by Roland Assaraf.

## 1 Introduction

A Fortran computer program produces *objects*, which are Fortran variables, of any type or dimension (scalar/array), containing quantities that we are after. For example, `psi` may be an object containing the value of a wave function evaluated at some electron coordinates. This object `psi` is constructed from other objects. For example, for the case of a Jastrow $\times$ single determinant wave function, we have `psi = jastrow * determinant` where `jastrow` and `determinant` are the values of the Jastrow factor and of the Slater determinant at the same electron coordinates. The later objects are themselves constructed from yet other objects. For example, `determinant` may be constructed from `orbitals` which is an array containing the values of the orbitals at the considered electron coordinates. And so on.

Clearly, there are *dependencies* between these objects, in the sense that the construction of a given object requires that other objects have already been constructed. In other words, the order in which the objects are constructed is important. Usually, in Fortran programs, these dependencies are *not* made explicit, and making sure that the order of the construction of the objects is correct is left to the programmer. In large codes, this may make difficult the implementation of a new object.

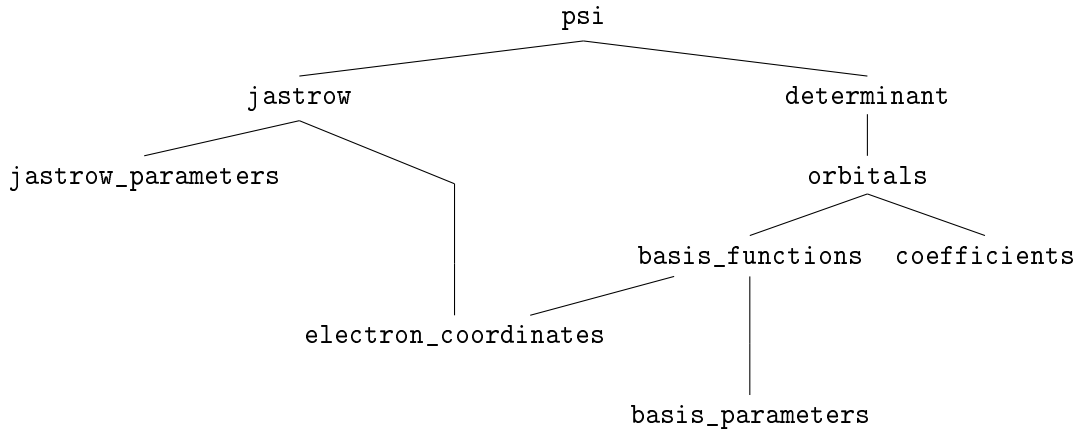
In the programming paradigm used here, the dependencies between the objects are made explicit. The programmer then does not need to take care of the order of construction of the objects. This facilitates the implementation of new objects.

## 2 A simple example

The programming paradigm is best explained on a simple example.

## 2.1 Dependency tree

Consider again the object `psi`, constructed from two objects `jastrow` and `determinant`, themselves constructed from other objects according to the following *dependency tree*:




Note that, strictly speaking, this is not a tree but a graph since the branches are not necessarily disjoint. For example, in the example above, the object `electron_coordinates` connects the two branches. We will nevertheless use the vocabulary of computer trees.

This tree encodes the dependencies between the objects. For example, `psi` depends on both `jastrow` and `determinant`. In turn, `jastrow` depends on `jastrow_parameters` and `electron_coordinates`. And so on. We will say that `psi` is a *child* of `jastrow` and `determinant`, and `jastrow` and `determinant` are the *parents* of `psi`. The objects without any parents (`jastrow_parameters`, `electron_coordinates`, `basis_parameters`, `coefficients`) are the *leaves* of the tree. These generally correspond to input parameters which must be given by the user when running a calculation.

The practical implementation of the dependency tree in the program will be explained later. For the moment, let us describe how calculations are done, assuming that the information about the dependency tree is available in the program.

## 2.2 Building subroutines

Each object which is not a leaf of the tree has a *building subroutine* which constructs it. So, written schematically, we have the five following building subroutines in the program:

```
subroutine psi_bld
psi = jastrow * determinant
end subroutine psi_bld

subroutine jastrow_bld
jastrow = f(jastrow_parameters,electron_coordinates)
end subroutine jastrow_bld

subroutine determinant_bld
determinant = f(orbitals)
end subroutine determinant_bld

subroutine orbitals_bld
orbitals = f(basis_functions,coefficients)
```

```
end subroutine orbitals_bld

subroutine basis_functions_bld
basis_functions = f(electron_coordinates,basis_parameters)
end subroutine basis_functions_bld
```

where, to simplify, we have shown the explicit expression of `psi` only, the expressions of the other objects are just written as (complicated) functions $f$.

The most part of the program is made of building subroutines, but they do *not* make *all* the program. Here are examples of subroutines which are *not* building subroutines:

- The subroutines reading the input. This is usually where the leaf objects are created.
- The subroutines writing the output.
- The subroutines containing the main iterative algorithms, such as the Monte Carlo algorithm or the wave-function optimization algorithm.

These subroutines manipulates the objects of the dependency tree with the two main commands which are explained in the next sections: `call object_provide` and `call object_modified`.

### 2.3 call object_provide

Suppose that the programmer wants to print out the value of `psi` at some place in the code (outside the building subroutines). He just needs to write the following lines:

```
call object_provide('psi')
write(6,*) 'psi=',psi
```

The instruction `call object_provide('psi')` does the following:
It checks if the object `psi` is *valid*, i.e. if it has already been calculated and can be used.

- If yes, then nothing is done.
- If no, then the program checks if its parents `jastrow` and `determinant` are valid.
  - If both are valid, then the program calls the building subroutine `psi_bld` and marks `psi` as valid.
  - If, for instance, only `determinant` is not valid, then the program checks if its parent `orbitals` is valid. If `orbitals` is valid, then the program calls the building subroutine `determinant_bld`, marks `determinant` as valid, then calls the building subroutine `psi_bld` and marks `psi` as valid. If `orbitals` is not valid, the programs checks its parents, and so on.

In other words, `object_provide('psi')` goes down recursively the dependency tree under `psi` until it finds valid objects. It then climbs up the dependency tree, constructing the objects one after the other, in the correct order, until it finally constructs `psi`.

In the case where `object_provide('psi')` goes down to a leaf object of the dependency tree (for example, `basis_parameters`) which is *not* valid, then it gives an error message indicating that this leaf object is necessary to construct `psi` but does not know how to construct this leaf object. Indeed, since a leaf object does not have a building subroutine, the program does not

know how to construct it. Leaf objects are instead usually read in from the input file at the beginning of execution and marked as valid then.

In summary, `call object_provide('psi')` has several advantages:

- It is *simple*. The programmer just needs to know the name of the object that he wants, here 'psi'. He does not need to know how this object is calculated by the program. He does not need to know about intermediate objects such as `orbitals`.
- It is *safe*. If a necessary leaf object is not available, the program will properly stops and explains what is missing.
- It is *efficient*, in the sense that this mechanism ensures that only what is needed is calculated, nothing more.

### 2.4 call object_modified

Another important ingredient remains to be explained. What if the value of an object, say `electron_coordinates`, is modified? Then, we need to make sure that if we need any child or grandchild object of `electron_coordinates`, this object must be recalculated with the new value of `electron_coordinates`. This is done by inserting, just after modifying `electron_coordinates`, the instruction `call object_modified('electron_coordinates')`:

```
electron_coordinates = (-2.1, 0.7, 1.5)
call object_modified('electron_coordinates')
```

The instruction `call object_modified('electron_coordinates')` marks `electron_coordinates` as valid, and recursively climbs up the dependency tree to mark as *invalid* all the objects depending on `electron_coordinates`, namely `jastrow`, `basis_functions`, `orbitals`, `determinant`, `psi`.

Thus, if any one of these objects is asked for afterwards, it will be recalculated with the new value of `electron_coordinates`. This is a *safe* mechanism since it prevents the programmer from forgetting to update objects in an iterative algorithm, a frequent bug otherwise.

Note that a typical use of `call object_modified` is after reading in leaf objects from the input to mark them as valid.

### 2.5 Implementation of the dependency tree

It was chosen to give the dependencies between the objects directly in the building subroutines. The advantage of this is that all the information about a given object is localized in its building subroutine, and not delocalized in different places in the code. Each building subroutine contains a *header* part where the object created by this building subroutine is indicated by `call object_create('...')` and all the parent objects are listed by `call object_needed('...')`. Thus, the building subroutines of our simple example actually look like:

```
subroutine psi_bld
if(header) then
   call object_create('psi')
   call object_needed('jastrow')
   call object_needed('determinant')
   return
endif
```

```
psi = jastrow * determinant
end subroutine psi_bld

subroutine jastrow_bld
if(header) then
   call object_create('jastrow')
   call object_needed('jastrow_parameters')
   call object_needed('electron_coordinates')
   return
endif
jastrow = f(jastrow_parameters,electron_coordinates)
end subroutine jastrow_bld

subroutine determinant_bld
if(header) then
   call object_create('determinant')
   call object_needed('orbitals')
   return
endif
determinant = f(orbitals)
end subroutine determinant_bld

subroutine orbitals_bld
if(header) then
   call object_create('orbitals')
   call object_needed('basis_functions')
   call object_needed('coefficients')
   return
endif
orbitals = f(basis_functions,coefficients)
end subroutine orbitals_bld

subroutine basis_functions_bld
if(header) then
   call object_create('basis_functions')
   call object_needed('electron_coordinates')
   call object_needed('basis_parameters')
   return
endif
basis_functions = f(electron_coordinates,basis_parameters)
end subroutine basis_functions_bld
```

For all the building subroutines, the header part is executed only once at the beginning of the execution of the program in order to construct the dependency tree. Then, in the actual calculations we always have `header=.false.` so the header part is bypassed.

# 3 More details

## 3.1 A building subroutine can create more than one object

Often, it is convenient to construct in the same building subroutine several objects. A given building subroutine thus generally creates several objects. For example, if the building subroutine `jastrow_bld` creates the two object `jastrow` and `jastrow_gradient`, it will look like:

```
subroutine jastrow_bld
if(header) then
   call object_create('jastrow')
   call object_create('jastrow_gradient')
   call object_needed('jastrow_parameters')
   call object_needed('electron_coordinates')
   return
endif
jastrow = f(jastrow_parameters,electron_coordinates)
jastrow_gradient = g(jastrow_parameters,electron_coordinates)
end subroutine jastrow_bld
```

With this extension, the nodes of the dependency tree are no longer the objects but instead the building subroutines, each one creating a list of objects. This extension does not cause any problem.

## 3.2 Dynamic allocations inside the building subroutines

If an object is an array, it is dynamically allocated inside its building subroutine, using `call object_alloc`. For example:

```
call object_alloc ('orbitals', orbitals, norb)
do i=1,norb
   orbitals(i) = ...
enddo
```

Note that the dimension `norb` itself is usually an object handled by the dependency tree. If it changes during the calculation, the array will be reallocated with the new size.

## 3.3 Indexes of objects

The functions `object_modified('...')` and `object_provide('...')` take as argument the name of an object given as a string. The name of the object is then looked up in an array which can take some time. If these functions are called in a part of the code that is very often executed, this is not efficient. In this case, we use directly the index of the object in the array, instead of its name. For example:

```
call object_modified_by_index (electron_coordinates_index)

call object_provide_by_index (psi_index)
```

### 3.4 Averages

```
call object_average_request ('dpsi_av')

call object_average_define ('dpsi', 'dpsi_av')
```

### 3.5 Nodes

```
object_provide_in_node(lhere, 'xi_en')
```

### 3.6 Interface with the old Fortran 77 part

The objects created in the old Fortran 77 files do not have building subroutines. If they are not available, the program does not know how to construct them. Nevertheless, to facilitate their use in the Fortran 90 part, they can be added as leaf objects of the dependency tree. For this, we just need to add `call object_modified('...')`, for example in the line just after a Fortran 77 object has been constructed. This provides a safe mechanism for using them, making sure that we use them only when they have been already calculated. In fact, I found this extremely useful when I started to program in CHAMP.

When this is judged useful, a Fortran 77 object can be converted to a Fortran 90 object, i.e. we write a proper building subroutine for it and it is then a normal object of the dependency tree. The Fortran 77 part of the code can then be progressively evolved in the style of the Fortran 90 part.