Combining Object-Oriented and Procedural Programming in Software

Wilfried Abels, TU Hamburg Harburg, Hamburg/Germany, w.abels@tu-harburg.de

Abstract

Large engineering software packages in procedural languages reach complexity limits for further development. Complete reimplemention in modern object-oriented languages is often financially not feasible. Problems and solution approaches are discussed using naval architectural design software as demonstrator. Concrete examples for the implementation of modern object-oriented elements into classical FORTRAN environments are given.

1. Introduction

The software package E4 is a ship design tool. Much know-how of hydrodynamic, manoeuvring, and other aspects of ship design has been implemented to this software in the last 15 years. Many people have built tools for different aspect of the ship design process, resulting in a heterogeneous software structure. The starting point for this paper was the need to redevelop an import module for measured wake fields and run interpolations on this imported data. A whole database for this information had been already implemented. There were methods for storage and handling wake fields, but not for altering them. This became problematic, because a new data format stored the supporting points no longer in equidistant intervals, which was necessary for the database of wake files and all other methods in E4 using wake fields. Modifications of the kernel of wake field storage was not an option. This kernel had been implemented traditionally with common blocks and functions with big parameter lists for data exchange. The orientation in this functionality needs at lot of time.

Starting from this, it was desirable to have a simple interface handling all communication with the storage mechanisms in the background. A software library called COSMOS implements the main database. The database allows to store information in hierarchical tables, making it a quasi-relational database. This database is used to store and exchange data with the total design process of a ship. If a software module of E4 processes this information the data is loaded from the COSMOS database and stored locally in FORTRAN common blocks. A problem by using such a common block is, that only one wake field can be handled at the same time. But for interpolation tasks, it is useful to have old and new versions in parallel. This kind of data storage implies a set of functions for wake field handling requiring very big parameter lists. Because the common block storage mechanism allows only one instance of wake field, all changes have to be done on this single data structure. If parts of information are necessary for later calculations, they have to be stored in different data structures.

This problem could be easily handled in an object-oriented model. A class \textit{Wakefield} can store all information needed for a wake field. Such a class could also encapsulate all necessary communication with the database layer. All common-block variables needed by old functionality can be stored in class attributes. If old functionality should be used the state of all variables has to be copied to the common-block data structure. Afterwards the needed calls of the kernel functionality are executed and at the end the total state of the common-block structure is transferred back to the attribute-list of the new-implemented object. A disadvantage of this approach is the big effort for data transfer. How could one connect the concepts of traditional procedural implementation with modern object-oriented designs in the same software project? To answer this question, the two concepts are compared considering an simple physical problem as example.

2. Procedural versus Object-Orientated Software Design

Traditionally, software design is procedural. The functionality of software is structured in a main program calling many functions and procedures. Beside the definition of functionality, the data can be
structured by its own. Most software languages include elements to define data structures for different data types, for arrays and for mixed data-structures. In this way you start to analyse your problem. The necessary information has to be structured and modules connected. Then algorithms determine how to process data. Functionality and data-description are divided in two parts. For numerical tasks there is no problem and it is a useful concept. A typical task for such a concept is the evaluation of mathematical equations.

The object-oriented design is a concept for modelling interactions between different elements. The evaluation of equations is not the main focus of this concept. The definition of data structures and functionality is connected to classes. To explain these differences, the following theoretical problem will be used: The motion of a planet system around the sun should be simulated. This is a simple example, because the only interaction between the elements is gravitation. The problem is analysed using Newtonian mechanics.

2.1. Mathematical Analysis

First, the physical problem is posed in terms of mathematical equations. The problem can be modelled as a set of points located in a 3-d environment. Each point has a mass and a velocity vector. High-school physics then allows to calculate the gravitation force between two objects:

\[ F_{i \rightarrow j} = \gamma \frac{m_i \cdot m_j}{|\vec{r}_j - \vec{r}_i|^3} \cdot (\vec{r}_j - \vec{r}_i) \quad ; \quad \sum_{j=1}^{N} F_{i \rightarrow j} = 0 \quad \text{and} \quad \vec{P}_i = \sum_{j=1}^{N} \vec{F}_{ij} \]  

With given forces, positions and velocities are easily updated:

\[ \vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{v}_i(t) \cdot \Delta t + \frac{\vec{F}_i(t)}{2 \cdot m_i} \cdot \Delta t^2 \quad \text{and} \quad \vec{v}_i(t + \Delta t) = \vec{v}_i(t) + \frac{\vec{F}_i(t)}{m_i} \cdot \Delta t \]  

These equations suffice for our example. They describe the motion of a set of planets. The following step translates this mathematical description into an algorithmic description.
2.1. Procedural Description

A classical approach to model this problem defines the needed data structure, e.g. in FORTRAN77:

```
Integer, parameter MAX_DATA=100

Integer N
Real, dimension(MAX_DATA)  x0, x1, y0, y1, z0, z1,
+                   vx0, vx1, vy0, vy1, vz0, vz1, m
common /planets/             x0, x1, y0, y1, z0, z1,
+                   vx0, vx1, vy0, vy1, vz0, vz1, m, N
```

All variables are defined as arrays and the maximal number of elements is coded as a hard parameter. The versions of 0 and 1 of all variables are used for the calculation from $t \to t + \Delta t$:

$$
\vec{r}_i(t) = \begin{pmatrix} x0[i] \\ y0[i] \\ z0[i] \end{pmatrix}; \quad \vec{r}_{i,\text{next}} = \vec{r}_i(t + \Delta t) = \begin{pmatrix} x1[i] \\ y1[i] \\ z1[i] \end{pmatrix}; \quad \vec{v}_i(t) = \begin{pmatrix} vx0[i] \\ vy0[i] \\ vz0[i] \end{pmatrix}; \quad \vec{v}_{i,\text{next}}(t + \Delta t) = \begin{pmatrix} vx1[i] \\ vy1[i] \\ vz1[i] \end{pmatrix}
$$

The next step would be the definition of functionality. The interesting problem is the calculation of the next time step. Traditionally a structure chart is used:

This is a classical efficient approach for the problem. You have a database and procedures for working on it.
2.2. Object-Oriented Description

The same problem can be modelled object-oriented, starting point from the class diagram:

In this case the problem is defined as a class for a planetary system containing a list of elements from the class MassElement. The class diagram shows all attributes and methods of an element. The philosophy would be now: An object from the class PlanetarySystem can be generated. Afterwards a number of objects from MassElement can be added to the object from PlanetarySystem. From this moment the software engineer can communicate with the numerical part by using the method PlanetarySystem::calcTimeStep(real: delta_time). The implementation and data-management of all elements are not longer of interest. Someone wanting to use the functionality of PlanetarySystem only needs the interface.

The implementation of the functionality is a little bit longer than as in the procedural case, but easy to understand because of clear interfaces. The main functionality of PlanetarySystem is the calculation of a time step:

```csharp
Void: PlanetarySystem:: calcTimestep (real: dt)
{
    foreach p element of self \rightarrow AllElements
    {
        p \rightarrow calc NextTimestep (dt)
    }
    foreach p element of self \rightarrow AllElements
    {
        p \rightarrow activate NextTimestep (dt)
    }
}
```

This method needs two methods from MassElement which can be realised as follows:
After all new values are calculated they have to be stored to the actual values by using:

**Void: MassElement:: calcNextTimestep (real: dt) {**

\[
\vec{F} := \vec{0}
\]

foreach p element of self \rightarrow \text{cont} \rightarrow \text{AllElements} { 
\vec{F} = \vec{F} + \text{self} \rightarrow \text{calcForce}(p)
}

self \rightarrow \vec{r}_{\text{next}} := \text{self} \rightarrow \vec{r} + \text{self} \rightarrow \vec{v} \cdot dt + \frac{\vec{F}}{2 \cdot \text{self} \rightarrow m} \cdot dt^2

self \rightarrow \vec{v}_{\text{next}} := \text{self} \rightarrow \vec{v} + \frac{\vec{F}}{\text{self} \rightarrow m} \cdot dt

**}
2.3. Comparison of both Approaches

The great advantage of the object-oriented approach is the small and simple interface. A software engineer, who only wants to use the module, has only a set of few interfaces to learn. Moreover, it is easy to implement more functionality to the model, without changing the existing functionality. Perhaps it is necessary to distinguish different types of MassElements in different types. For example, it is then possible to build subclasses for planets, moons, and stars:

![Diagram of MassElements]

A second advantage for object-oriented modelling is that many structures for typical problems are already developed, e.g. Gamma (2004). But there are also disadvantages. The numerical part of this approach is less efficient. Because of the partitioning in the two classes there are a lot of references between all objects. Each access to a value needs a tracing of all relevant references. Because these object references are within the numerical loops they create performance bottlenecks. A second disadvantage is the need of a more complex declaration for all functionality. A software engineer needs normally more time to implement the same functionality. This may be offset by time-savings later when modules are reused.

3. Computer Languages in Combination with Design Philosophies

There is much debate about which language is better for a given problem. Mostly there are two statements:

1. If you want to implement modern object-oriented designs concepts you have to use an object-oriented language like C++ or Java.
2. An optimising compiler for classical FORTRAN is more efficient than compilers for high-level languages

The object-oriented design philosophy is based on the concept of encapsulation of data and functions to an object. Each object contains a set of values and a set of functions. It is possible to generate a set of objects from the same class. A class is the analogue to a type definition in procedural languages, with the main difference that a class defines functionality further. In the same way dynamic variables are generated, an object (called instance of a class) is generated by a new-operation. Every instance of a class knows its own variables, which are independent from the variables of other instances of the same class. Further, an important feature of object-oriented design philosophy is the inheritance. This allows building subclasses which include all properties of the parent class. Object-oriented languages include features to implement this concept directly, allowing to define classes containing variables and functions. Moreover, a dynamic memory management is available for generating a set of instances. Classical languages like FORTRAN do not contain keywords for defining classes. A further problem is that the object-oriented design concept needs the possibility to generate and destroy
objects during runtime. For example, FORTRAN77 has no features for a dynamic memory management. Is it therefore impossible to write object-oriented software in FORTRAN? Clearly no. Every software language is compiled to machine language. Thus all languages share some common ground, however big their differences may be. A problem that can be described in one language, can be described in every other language. Sometimes this is not easy, but it is always possible.

The second point of performance has to be discussed, too. Do FORTRAN programs perform better than C++/Java programs. There are real performance differences between different compilers. Different compilers have different modules for optimising a program. Therefore a compiler from vendor A may give other results than a compiler from vendor B for the same source code. These are differences in the implementation of a compiler for one language. But are there theoretical differences in performance between different computer languages? There is no short and simple answer. The same problem can be modelled in different ways, see the example in Ch.2. Normally, software engineers would take a procedural language (perhaps FORTRAN) to implement the procedural model and an object-oriented language (C++/Java) to implement the object-oriented model. Afterwards they would do performance test with both versions. The result would be that the procedural version performs better. What is the meaning of this result? Is this true that FORTRAN programs have more performance than C++/Java programs? For an answer it is necessary to analyse the different way both versions are working. The procedural one contains only two loops which run over a fixed array of values. Start and end of the loops are clear. Therefore, a compiler can build a very optimised code. The compiler has all information about the loops and about the memory arrangement of the needed values. In contrast to this, the object-oriented version is much more flexible. But this flexibility means much more overhead. The compiler does not know anymore the exact memory arrangement and the exact process flow over the calculation loops. Consequently, the compiler has to include code to collect this information during runtime. For example the access to an element of a hard-defined array needs only one clock cycle if you have the address within a register of the CPU. But if the CPU only knows a pointer to the memory of an object it is necessary to calculate the exact position of the needed value by adding an offset to the base-address of the object. This needs a second clock cycle. One clock-cycle is not very much, but if this operation is repeated very often in a loop it can mean a bottleneck in performance. This is a typical example, where a drop in performance does not result from different compilers, but directly from different modelling concepts. An honest comparison between compilers for different languages needs a source code which models the problem in the same way. A comparison of C++ and Java (both object-oriented languages) results in a further important point for this discussion. Normally, a C++ program has better performance. But even if the optimisation features and the software model are same, the main difference is that C++ is compiled directly to the hardware language of the used CPU. In contrast Java is compiled to a hardware independent byte code which has to be emulated on the real processing unit. This method is useful if software should be run on different platforms without compiling it anew in every case. Applications run on the Internet use this feature. A server can collect a set of software and clients can load this software and execute it on a runtime environment. All hardware dependent properties are hidden within this environment. This big increase in flexibility comes at the price of decreased performance. But this paper will only focus on the aspects of modelling.

In conclusion, the performance of software is independent of a computer language. Differences result from the optimisation module of a compiler and from the design model. In principle, it is possible to implement procedural designs in object-oriented languages and object-oriented designs can be implemented in procedural languages.

3.1. Implementation of Object-Oriented Designs in Classical FORTRAN Environments

Why would we implement object-oriented designs in a FORTRAN environment, when there are better suited languages available? To implement the same design in FORTRAN needs much more effort. Many construct directly available in object-oriented languages have to be programmed manually in FORTRAN. If we start from scratch there would be no sense in implementing object-oriented structures in a procedural language. In this case, an object-oriented language would be used
and the software engineer could decide in which parts of the project object-oriented flexibility is useful and in which not. But in many cases a project does not start from scratch. Developments of many years cannot be implemented anew. No one would pay for all the needed man-years. Such external constraints motivate the hybrid object-oriented approach in FORTRAN outlined below.

3.2. Simple object definition in FORTRAN77

If you only need the possibility to encapsulate data from the residual program, this is possible by the use of the save-parameter:

```fortran
Subroutine xyz_new()
  Real : a,b,c
  Save
    A=0.0
    B=0.0
    C=0.0
  return
Entry xyz_delete()
  ...
  return
entry xyz_calc(u,v,w)
  ...
  return
end subroutine
```

Even this simple source-code example can be interpreted as an object. There is a set of attributes (a,b,c of type real) and three methods working on these attributes (xyz_new, xyz_delete, xyz_calc). Because the variables are defined with the save attribute they are not destroyed at the end of a subroutine call. This is the same behaviour as in attribute variables of an object. Different methods of the object can be implemented by different entries. Because the entries are owned by the same subroutine, they all have access to the same variables like methods of the same class.

This interpretation of the given example is a very simple type of objects. The main disadvantage of this approach is the impossibility of generating more than one instance of a class. But if a software design only needs one instance of a class, this is a useful solution.

3.3. Approach for Multiple Class Instances

To implement classes allowing more than one instance of objects requires a memory management to handle different sets of values for each class instance. One could define a new data type containing all attribute variables of the needed class. For example, the class `MyClass` needs one integer and one real. In FORTRAN90 it is possible to collect this values to one type:

```fortran
type MyClass_type
  integer :: i
  real    :: a
end type MyClass_type
```

Now an array with elements of this type can be build:

```fortran
Type (MyClass_type), dimension(MAX_OBJECTS) :: MyClass_data
```

This is an easy way to handle memory for a set of objects. An additional array can control which data sets are in use.
Logical, dimension(MAX_OBJECTS) :: MyClass_isUsed

At program start the whole array has to be initialised with .FALSE. Later every new instance has to search for a free position. The corresponding array index is the pointer to the object. The new-method of MyClass could be implemented like:

Function MyClass_new()
    Integer :: MyClass_new
    Integer :: I
    Logical :: ok
    Ok = .false.
    Do I=1,MAX_OBJECTS,1
        If (MyClass_isUsed .eq. .false) then
            MyClass_isUsed[I] = .true.
            MyClass_data[I]%i = 0
            MyClass_data[I]%a = 1.0
            MyClass_new = I
            Return
        End if
    End do
    MyClass_new = -1
    Return
End function MyClass_new

Each further method of the class MyClass needs only the integer index returned by the new-method to access its own variables. If an object is no longer used a delete-method can free the used memory:

Procedure MyClass_delete(id)
    Integer :: id
    MyClass_isUsed[id] = .false.
End procedure MyClass_delete

In the same way each method of class MyClass has access to its own attributes. Each access has to be written like:

MyClass_dat[id]%<attribute_name> = …

If the software engineer accepts this convention of implementation there is no restriction in implementing an object-oriented design.

3.4. Class Inheritance by Object Encapsulation

In an object-oriented language the definition of inheritance is easy. One statement in the header of a class to specify the parent class suffices. This feature is not available in procedural languages. But in the same way a class can inherit functionality, it is also possible to include one instance of a parent object within a child object. In this case all functionality handled by the parent is only transferred to the encapsulated parent object. These two concepts are equivalent. An implementation example is:

type ChildClass_type
    integer :: parent_object
    real :: c,d
end type ChildClass_type
The following line has to be added to the new-method:

```plaintext
ChildClass_data[id]%parent_object = ParentClass_new()
```

And the delete-method needs:

```plaintext
Call ParentClass_delete(ChildClass_data[id]%parent_object)
```

Finally, each method to be used from the parent class must be implemented as follows:

```plaintext
Procedure ChildClass_<name>(id, ...)
   Integer :: id
   Call ParentClass_<name>(ChildClass_data[id]%parent, ...)
End procedure ChildClass_<name>
```

A little problem there is if a type casting between child and parent objects is necessary. Perhaps you have a method needing an object from type of the parent class, but you have generated a child object. In object-oriented software, a method capable of handling a parent class automatically can handle all subclasses. In the above shown example this is not directly possible. A type control is not included to the shown concept. If you put the integer index of the child object to a method for the parent object, the whole memory management would be corrupted because all memory access would be done on the wrong data array. The only way to solve this problem would be within an explicit object-identifier translation. The child class has to implement such a method:

```plaintext
function ChildClass_castToParent(id)
   integer :: ChildClass_castToParent
   ChildClass_castToParent = ChildClass_data[id]%parent
   Return
End function ChildClass_castToParent
```

The situation is more complicate from the side of the parent object. This casting concept is only useful if it would be possible to cast an object later back to the child class. But for such a method the parent object has to know whether it is part of a subclass or not. This is not impossible, but two new methods are needed within the parent class:

```plaintext
function ParentClass_castToChild(id)
   integer :: ParentClass_castToChild
   ParentClass_castToChild = ParentClass_data[id]%child
   Return
End function ChildClass_castToParent
```

```plaintext
Function ParentClass_newFromChild(child_id)
   Integer :: ParentClass_newFromChild
   Integer :: child_id
   Integer :: id
   Id = ParentClass_new()
   ParentClass_data[id]%child = child_id
   ParentClass_newFromChild = id
   Return
End function ParentClass_newFromChild
```

Now the new-method of the child class has only to call the newFromChild-method of the parent class instead of the original new-method. This concept implements only single inheritance, but could be extended to multiple inheritance. But this will not be a topic of this paper.
3.5. Implementation of Classes with Virtual Methods

Another important aspect of object-oriented designs is the definition of virtual methods in classes. A virtual method is used to define a function in a parent class to be implemented in every subclass. The parent can implement this method, but does not have to do it. Only the interface of this method has to be declared. This concept allows the implementation of classes capable of handling objects in a general way. Perhaps a method needs the displacement of a ship. In the system there are many classes describing different types of ship. Every ship needs a different way to calculate the displacement. But someone who implements a method which only needs the displacement, does not want to know the way every special ship calculates this value. This software engineer wants to give a function which returns only this value and the way of calculation is not in his focus. He likes to write such a source code line:

\[ D = \text{GeneralShip\_getDisplacement}(\text{ship\_id}) \]

In this case there has to be a parent class GeneralShip with the method getDisplacement. The problem is that the class GeneralShip does not know how to calculate the displacement. Then the object is not generated directly from the class GeneralShip, but from a specialised subclass, perhaps FerryShip. In an object-oriented language the runtime system knows which specialised class is behind a call to a virtual method of a parent class. A procedural language does not support this. Therefore, you have to implement the needed feature yourself. The problem is that now the parent class has to handle the information for the subclasses. The parent class object has to store information during runtime about the subclass it has been generated from. Type information is necessary. Implementing a global unique integer-parameter can be done like:

```fortran
Integer,parameter :: GeneralShip\_classNo = 1
Integer,parameter :: FerryShip\_classNo   = 2
Integer,parameter :: TankShip\_classNo    = 3
...
```

These parameters could be collected in a global module which has to be included to the whole software package. Next, the parent class has to manage this information. The attribute definition could be implemented like:

```fortran
type GeneralShip\_type
   integer :: child\_object
   integer :: child\_classNo
   ...
end type GeneralShip\_type
```

Now the implementation of GeneralShip\_getDisplacement is easy to handle:

```fortran
Function GeneralShip\_getDisplacement(id)
   Real :: GeneralShip\_getDisplacement
   Integer :: child
   Real    :: d
   Child = GeneralShip\_data[id]\%child\_object
   Select case(GeneralShip\_data[id]\%child\_classNo)
      Case(FerryShip\_classNo)
         d = FerryShip\_getDisplacement(child)
      Case(TankShip\_classNo)
         d = TankShip\_getDisplacement(child)
      Case default
         d = -1.
   End select
End function GeneralShip\_getDisplacement
```
End select

GeneralShip_getDisplacement = d
Return
End function

Now the software engineer can implement for every subclass separately the calculation for the displacement. Another engineer can handle a problem with ships in a general way, independently. It is possible to build a design model of the software with clear borders between different modules.

4. Conclusions

Big software systems grow over many years. The design concepts change rapidly, because software engineering is a young science. At the beginning of a project it is impossible to know all relevant parameters for the future. Therefore, past design decisions result in problems now. Further, the financial investments in such software systems are very high, making new implementations usually impossible. On the other hand, it is often not possible to implement more and more functionality, because the complexity of software designs has reached a critical limit. At this point, the software needs to be restructured (without a total new implementation), defining independent modules with clear tasks and interfaces. This allows an engineer to identify the relevant software parts if changes or extensions have to be made. Object-oriented design concepts were developed to support such modularisation.

While procedural languages like FORTRAN do not support object-oriented modelling, it is still possible to implement such designs. All unsupported concepts have to be implemented by the engineer himself. This is exhausting, but not very difficult. Moreover, the engineer has a bigger responsibility for the implementation, because the compiler does not control correctness of implementation of object-oriented models. Despite these short-comings, it is a passable way to use modern design concepts in old projects without completely re-implementing. A set of FORTRAN90 examples shown here can be used to implement concepts of class definitions, inheritance and virtual class methods. This allows to connect different design concepts in the same project, without mixing different programming languages.

References

GAMMA, E. (2004), Design patterns: elements of reusable object-oriented software Addison-Wesley professional computing series