# On Structured Representation of Physical Objects

Michal Mnuk[1] and Gerd Baumann[2]

[1] Institut für Informatik
Technische Universität München
D-80290 Munich, Germany
mnuk@in.tum.de
[2] Universität Ulm
Abteilung für Mathematische Physik
Albert-Einstein-Allee 11
D-89069 Ulm, Germany
Gerd.Baumann@physik.uni-ulm.de

**Abstract.** We present a tool for structured representation of physical objects in *Mathematica*. It is based on the concept of classes and hierarchies and is designed to maintain objects' properties and relations in a consistent, transparent and extensible manner.

## 1 Goals

Modeling is a sophisticated process of oscillation between reality and manageability. It eventually converges to a mathematical description which captures all aspects essential for a particular purpose and, at the same time, is simple enough to be treated by formal methods.

We start with an observation that every modeling process involves working with objects and its primary goal is to select features relevant to a specific problem together with relations between them. Different problems in a specific area usually involve a rather limited number of objects. The problem itself is then reflected in the relations between them.

From this point of view it is natural to build a knowledge base containing description of basic objects which often appear in modeling. Such a database would significantly speed up modeling and provide a basis for rapid prototyping. It is not only a tedious and error prone task to type in things over and over again. In many cases solutions for recurring problems are difficult to obtain although partial answers can be computed and stored in advance.

Our primary motivation is to facilitate the modeling process in engineering by providing a transparent and easy-to-use knowledge base of objects. In this paper we stipulate and analyze requirements on such a system and present a *Mathematica* package to store and retrieve information about physical objects.

The proposed solution is designed to be primarily static by nature. It is not supposed to include "automatic solvers" although internal and external algorithms will be utilized to perform computation whenever this appears appropriate. Extensions of the knowledge base will always be supervised. The reason is that vast majority of practical problems does not provide for easy answers. Even though today's computer systems became very powerful, automatically obtained solutions still need to be post-processed by experts. This makes it impossible to simply add them to an existing knowledge base since sooner or later its consistency would inevitably be destroyed.

## 2 Objects in Modeling

When we analyze requirements on the design of a system for object representation, we naturally come across principles already used for years in object oriented programming. It is not hard to see that same or very similar principles will govern the design here.

Object oriented programming introduced the concept of classes and inheritance into software engineering. Solutions often simplified considerably when a hierarchical structure was imposed on problems.

Another aspect of object oriented programming may be even more important. Classes and their objects are encapsulated, stand-alone and well specified entities which behave the same way in any program – a key property of reusable components. This makes it possible to publish code which can be reused without change in various settings.

It is clear that these paradigms apply well to our problem. A hierarchical structure is already in the nature of all mathematical and physical objects. The closeness of used entities is a prerequisite for the use of represented objects in different environments. These two concepts will play a primary role in the design of the knowledge base.

## 3 Object Oriented Methods in Computer Algebra

The fact that principles of object oriented programming are compatible with structures in scientific computing led to adoption of object oriented design in computer algebra. The combination of these two fields yields the key to a solution of our problem. On the one hand, object oriented design provides for encapsulation and reusability of objects, on the other hand, computer algebra offers a variety of algorithms to work with them.

However, up to now there is no single system that would contain these features in a well balanced symbiosis. We encounter either systems offering implementation of the class concept (C++, Java, SML, Aldor, ...), or single- or multi-purpose (computer algebra) systems providing more or less large set of algorithms (Maple, *Mathematica*, GAP).

Despite of this fact, lot of successful work was already done on using object oriented methods in computer algebra. There is an extensive literature on object oriented numerical methods ( [2, 4, 5, 14]). Numerical algorithms benefit from the fact that most of data types they work on are largely supported by classical programming languages so there are quite a few choices. However, numerics does not belong to areas which primarily benefit from object oriented methods.

Another approach integrates object oriented programming languages and computer algebra in one application ( [1, 6, 9, 11]). However, these solutions are mostly aimed at specialized fields.

The last option is to stay solely within a computer algebra system which needs to be extended in several ways. Maple was long time lacking a proper concept for data encapsulation. It was finally included in Version 7. An example of object oriented programming in numerical computation in Maple can be found e.g. in [8]. Despite of this, Maple's imperative nature is not well suited for complex manipulations on data and code which is necessary when implementing object oriented features.

Probably the first implementation of classes in *Mathematica* was done by R. Maeder in [10]. Even though this code was just a proof of concept, it demonstrated the strength of the language. *Mathematica* provides means for building structures and for data encapsulation, but this functionality is not powerful enough to directly map the concept of classes onto it.

Applications in engineering require an extensive collection of algorithms that is currently provided only by general computer algebra systems like Maple or *Mathematica*. In Section 5 we present the package *Elements* which goes beyond Maeder's work and which proved to be well suited for the purpose of building a knowledge base of objects. Now, we postulate and analyze requirements to be fulfilled by such a system.

## 4 Design Requirements

The package *Elements* is supposed to support and facilitate the modeling process by providing an efficient working environment. This is the primary goal which all other requirements specified below are emanating from.

Even though the focus has been put on the object oriented approach, we want to stress that that our intension is not to mimic paradigms found in other programming languages. The capabilities of *Mathematica* will be enhanced only to an extent that is necessary to achieve the specified goals.

### 4.1 Hierarchical Structure

We already argued that there is no apparent reason for a knowledge base to have a particular structure. But it is a fact that human beings try to organize things into categories. This gives rise to *hierarchies* which make relations between objects more comprehensible.

Similarly as in object oriented programming, the key role in our approach is played by *classes*. They represent different abstraction levels of objects. Classes at higher levels describe general principles which are "specialized" on the way down in the hierarchy. Subordinate classes inherit properties of their ancestors, they may restrict or modify them and/or introduce new ones. Classes serve as templates for creating objects. However, they will often be used on their own. We stipulate:

> *Classes are organized in hierarchies such that every class has exactly one ancestor (superclass). They serve as templates for creating objects.*

This principle enables us to provide an exact, concise and irredundant description of objects.

### 4.2 Reusability of Objects

One of our primary design goals is *reusability* of objects – the ability to be transparently used in different settings. This, at least, presumes that objects are represented in a way that makes them independent of their environment. The next stipulation is:

> *Represented objects can be easily imported and used in different environments. The behavior of objects does not depend on a specific environment.*

### 4.3 Soundness of Representation

When a programmer wants to use a piece of code written by someone else, he or she needs a clear picture of what this code expects as input and what it yields on output. While this can be rather easily fulfilled in classical programming languages, more care needs to be taken when dealing with complex objects in computer algebra. Usually, the input and output of functions implemented in programming languages is determined by the underlying type system which is in most cases quite simple.

A knowledge base built upon the package *Elements* will store models of objects, for example metal rods. Suppose, this representation will contain certain physical properties of rods together with a function describing the bending under force impace. As this information may depend on various assumptions which cannot be deduced from properties of a rod itself (like some simplifications done while solving the differential equation describing the bending), the use of the model is limited unless all restrictions are known to the user. Hence, they must be properly represented and stored with the modeled objects.

> *Represented objects are accompanied with a sufficient amount of information that makes validity checking of stored data possible.*

### 4.4 Consistency of Representation

It is obvious that introducing inconsistencies or contradictions into a system that stores knowledge makes it unusable. Hence, even if the soundness of the represented data is established, in the course of working with the data this state must be preserved. Hence:

> *The system provides suitable means to check consistency of stored data.*

In the ideal case, these means are used to automatically obtain statements about constraints and validity of new classes and objects. They ensure that subclasses are properly constructed and that they obey restrictions imposed by parent classes. If automatic validity checking is not possible or feasible, the user must be able extract enough information from object representation to perform this task by hand.

### 4.5 Data Exchange

We require that a knowledge base can be easily integrated into existing environments and must be capable of communicating the data with the outside world:

> *The knowledge base is able to import and export data and communicate with other systems using standard interfaces.*

This will make it possible to incorporate the stored information into other systems as an add-on for essentially no extra work. On the other hand, the knowledge base will benefit from any additional information the user may have available.

## 5 Elements – A Tool for Representing Objects in Mathematica

After we have specified basic requirements to be fulfilled, we present a brief survey on current capabilities of the package *Elements*. The first two subsections illustrate the implementation of standard notions of class and inheritance. Then we apply this machinery to the problem of damped harmonic oscillation.

### 5.1 Classes and Objects

A class is a structure consisting of *properties* and *methods*. Properties represent constant features, methods are usual functions. Properties and methods are kept inside of classes and they do not interfere with the environment. On the other hand, all objects in the current session (especially the complete functionality of *Mathematica*) are accessible from within classes.

There is a distinguished class `Element` which is the root of the whole hierarchy. New classes are derived from existing ones. Below, a new class `C1` (name is specified in the first argument) is derived from the class `Element`. It has properties `a` and `b` (there is one option `Note` associated with `b`) and a method `f` defined for numeric arguments.

```
In[]:= C1 = Class["C1", Class["Element"],
           (* Properties *)
           {a = 1,
           {b = π, Note → "Some transcendental number"}},
           (* Methods *)
           {f[x_?NumericQ] := b x}
         ]
Out[]= - Class C1 -
```

Classes and objects may call a number of implicitly defined methods:

```
In[]:= C1.Properties[]
Out[]= {a, b}

In[]:= C1.Methods[]
Out[]= {f}

In[]:= C1.BaseClass[]
Out[]= - Class Element -
```

New objects are created by the class method `new`.

```
In[]:= o1 = C1.new[]
Out[]= - Object of C1 -
```

They inherit all properties from its base class and can modify them to an extent specified in class' property declaration. If a property is declared to allow only positive values, no object may set the value of its own copy to something negative. Objects may also call all methods of its base class.

```
In[]:= o1.Properties[]
Out[]= {a, b}

In[]:= o1.a
       o1.b
Out[]= 1
Out[]= π

In[]:= o1.a = 5;
       o1.a
Out[]= 5

In[]:= o1.f[3]
Out[]= 3 π
```

It is considered an error if no method with a matching signature is found – it is meaningless for our purpose to return unevaluated calls.

```
In[]:= o1.f["string"]

Class :: "nometh": Method f with specified signature not found
```

### 5.2 Inheritance

Hierarchical structures, at least in software engineering, benefit from the ability to define properties and methods in one place and make them automatically available in other classes.

Below, a new class `C2` is created which inherits properties and methods from its base class `C1`. It declares a new property `a` which "shadows" the property `a` inherited from `C1`. It overloads the method `f` from `C1` by adding a definition with another signature. Among the new method `g`, it defines two *constructors*. They have the same name as the class itself and are called after the object was created by the class method `new`. Which constructor is invoked depends on parameters supplied to `new`.

```
In[]:= C2 = Class["C2", C1,
           {a = 2},
           { (* overloading f *)
             f[x_, y_] := a x + b y,
             g[x_?NumericQ] := a f[x],
             (* constructors *)
             C2[anew_?NumericQ, bnew_?NumericQ] := Block[{},
                 a = anew; b = bnew],
             C2[anew_, bnew_] := Block[{},
                 a = b = ∞]}
         ]
Out[]= - Class C2 -
```

The object `o2` is created by invoking the standard mechanism (method `new[]`). It inherits anything from its base class.

```
In[]:= o2 = C2.new[]
Out[]= - Object of C2 -

In[]:= o2.a
Out[]= 2

In[]:= o2.g[6]
Out[]= 12 π

In[]:= o2.f[1, 2]
Out[]= 2 + 2 π
```

The object `o2x` is created and immediately modified by the constructor `C2[anew_?NumericQ, bnew_?NumericQ]`.

```
In[]:= o2x = C2.new[4, 5]
Out[]= - Object of C2 -

In[]:= o2x.a
       o2x.b
Out[]= 4
Out[]= 5
```

### 5.3 Example – Damped Harmonic Oscillation

In this section we provide an example that illustrates the intended use of the package *Elements*. Instead of describing a real physical thing, this time we apply the object oriented approach to a differential equation for damped harmonic oscillation. This example will exhibit the same phenomena as any real physical object.

Suppose, we want to describe a simple one-dimensional motion of a body on a spring with damping. Let $m$ denote the mass of the body, $k$ the spring constant and $p(t)$ the damping function which should in general depend on the time $t$. The position $x(t)$ of the body in time is a real function defined on $[0, \infty)$ satisfying the following differential equation with initial values:

$$\begin{aligned} m\,x''(t) + p(t)\,x'(t) + k\,x(t) &= 0 \\ x(0) &= x_0 \\ x'(0) &= v_0. \end{aligned} \tag{1}$$

Depending on $p(t)$ it is more or less easy to obtain a solution. Let us assume that we are interested in the simplest case where the damping function is constant. Typing in (1) into *Mathematica* we obtain:

```
In[]:= DSolve[
         {m x''[t] + p x'[t] + k x[t] == 0, x'[0] == v0, x[0] == x0},
          x[t], t]
```

$$\text{Out[]= } \left\{\left\{x[t] \to \frac{e^{\frac{\left(-p-\sqrt{-4\,k\,m+p^2}\right)\,t}{2\,m}}\left(-2\,m\,v0+\left(-p+\sqrt{-4\,k\,m+p^2}\right)\,x0\right)}{2\sqrt{-4\,k\,m+p^2}} + \frac{e^{\frac{\left(-p+\sqrt{-4\,k\,m+p^2}\right)\,t}{2\,m}}\left(2\,m\,v0+\left(p+\sqrt{-4\,k\,m+p^2}\right)\,x0\right)}{2\sqrt{-4\,k\,m+p^2}}\right\}\right\}$$

However, this "solution" has several problems. The most serious one is the presence of the term $p^2 - 4km$ in the denominator. Moreover, the parameters $m$, $p$ and $k$ appear free in the expression. Setting some of them to $t$ yields unexpected results. Clearly, a solution in this form is incomplete and requires external information in order to be used in further computation.

Next, we show how the solution of Equation (1) can be encapsulated within an object using the package *Elements*. We declare a class `DHO` which represents a general solution to this problem (without restricting the damping function) and set this solution to the value `Undefined`.

```
In[]:= DHO = Class["DHO", Class[Element],
           {description = "Damped harmonic oscillation",
            equation = m x''[t] + p[t] x'[t] + k x[t] == 0,
            {m = 1, Description → "Mass",
                    Domain → Positive},
            {p = 0, Description → "Damping function"},
            {k = 0, Description → "Spring constant",
                    Domain → NonNegative},
            {x0 = 0, Description → "Initial displacement"},
            {v0 = 0, Description → "Initial velocity"}},
           {x[t_] = Undefined}
         ]
```

```
Out[]= - Class DHO -
```

To specify the same problem with a constant damping function, we introduce a new class DHOc which declares a new property p and restricts its domain of definition. Note that all other properties are inherited from the class DHO. In this case, a closed form of the solution exists and can be hard-coded in the class DHOc.

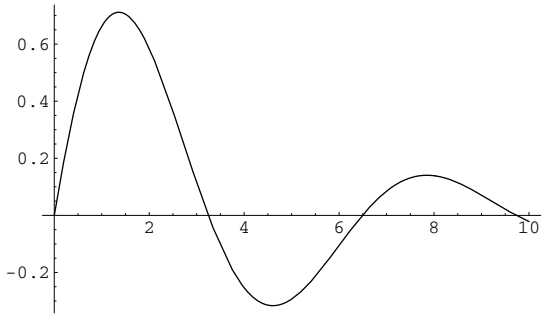
```
In[]:= DHOc = Class["DHOc", DHO,
          {description = "Oscillation with constant damping",
           equation = m x''[t] + p x'[t] + k x[t] == 0,
           {p = 0, Description → "Damping factor",
                  Domain → NumericQ}},
           {x[t_] := Block[{D = p^2 - 4 k m},
              Which[
                D == 0,
                  e^(-p/(2 m) t) (x0 + t (v0 + (p x0)/(2 m))),
                D > 0,
                  e^(-p/(2 m) t) (x0 Cosh[(t Sqrt[D])/(2 m)] + ((2 m v0 + p x0) Sinh[(t Sqrt[D])/(2 m)])/Sqrt[D]),
                D < 0,
                  e^(-p/(2 m) t) (x0 Cos[(t Sqrt[-D])/(2 m)] + ((2 m v0 + p x0) Sin[(t Sqrt[-D])/(2 m)])/Sqrt[-D])
              ]],
            plotDisplacement[start_, end_] :=
              Plot[N[x[t]], {t, start, end}],
            DHOc[mnew_, pnew_, knew_, x0new_, v0new_] :=
              Block[{},
                m = mnew; p = pnew; k = knew; x0 = x0new; v0 = v0new; ]
          }
         ]
Out[]= - Class DHOc -
```

This class represents a complete solution to the simplified problem which is clearly superior to that supplied by *Mathematica*'s DSolve. In addition, the class DHO is extended by a method to plot the displacement function. To solve the problem of damped oscillation for specific parameter values, we create an object from the class DHOc and set the parameters accordingly.

```
In[]:= body = DHOc.new[1, 1/2, 1, 0, 1]
Out[]= - Object of DHOc -

In[]:= body.x[t]
Out[]= (4 e^(-t/4) Sin[(Sqrt[15] t)/4])/Sqrt[15]

In[]:= body.plotDisplacement[0, 10]
```



```
Out[]= -Graphics-
```

And, obviously, the case where $p^2 = 4km$ is treated correctly now.

```
In[]:= bodyc = DHOc.new[1, 2, 1, 0, 1]
       bodyc.description = "Critical damped system";
Out[]= - Object of DHOc -

In[]:= bodyc.x[t]
Out[]= e^-t t
```

The benefit to the user is apparent. The method `x[t]` provided by the class `DHOc` yields the correct solution of (1) for any valid combination of parameters $m$, $p$ and $k$ which are now quantified by the object they belong to, i.e. they do not interfere with the variable $t$ nor with any other object. An attempt to set them to illegal values will be immediately recognized by the type system. In this way, the user obtains a solution which is general, reliable, correct, easy to use, and containing all information needed to guarantee its validity.

## 6 Further Development

At the time of this writing the package *Elements* implements the functionality discussed in Sections 4.1, 4.2 and 4.3.

Concerning the type system, we have developed several strategies for implementing it. In our opinion, types are inevitable for maintaining the consistency of stored information. However, it is a delicate matter to find a balance in this issue. Typing must be strong enough to allow of exact reasoning. On the other hand, it must not become an obstacle to the expressivity. Type information is stored in attributes of properties and is optional. The exact handling of types will be settled later, after a careful analysis of practical needs.

Another issue which needs further consideration is the communication of the knowledge base with its environment. There are several possibilities available – OpenMath [3], CORBA [12,13], MathLink and JLink, and others. Which concept will eventually be used, largely depends on its flexibility to accommodate to varying needs.

## 7 Conclusions

In this paper we discussed basic guidelines for building a knowledge base of objects. We presented the package *Elements* written in *Mathematica* that implements the core functionality of the system. We showed that combining the computational power of computer algebra software with clear structures and proper type checking provides a firm foundation for speeding up the work flow in modeling processes. The soundness and consistency of stored information ensures its practical applicability in a variety of settings. The ability of the knowledge base to communicate with its environment facilitates its use within an existing infrastructure.

## 8 Acknowledgments

The authors would like to thank an anonymous referee who substantially contributed to the improvement of the presentation of this paper.

## References

1. Matthias Berth, Frank-Michael Moser, and Arrigo Triulzi. Implementing computational services based on OpenMath. In Ganzha et al. [7], pages 49–60.
2. Guido Buzzi-Ferraris. *Scientific C++: Building numerical libraries the object-oriented way.* Addison-Wesley, 1993.

3. O. Caprotti and A. M Cohen. The OpenMath standard. Technical report, The OpenMath Consorcium, http://www.openmath.org/, February 1999.
4. Kevin Dowd. *High Performance Computing*. O'Reilly & Associates, Inc., 1993.
5. Paul F. Dubois. *Object technology for scientific computing*. Prentice Hall, 1996.
6. Peter Fritzson, Johan Gunnarson, and Mats Jirstrand. MathModelica. an extensible modeling and simulation environment with integrated graphics and literate programming. In Martin Otter, editor, *The proceedings of the 2nd International Modelica Conference, Oberpfaffenhofen, Germany*, pages 41–54. DLR, Oberpffafenhofen, Germany, March 2002.
7. V.G. Ganzha, E.W. Mayr, and E.V. Vorozhtsov, editors. *Computer Algebra in Scientific Computing, CASC 2000. Proceedings of the Third International Workshop on Computer Algebra in Scientific Computing, Samarkand*. Springer-Verlag, October 2000.
8. Victor G. Ganzha, Dmytro Chibisov, and Evgenii V. Vorozhtsov. GROOME – tool supported graphical object oriented modelling for computer algebra and scientific computing. In V.G. Ganzha, E.W. Mayr, and E.V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing, CASC 2001. Proceedings of the Fourth International Workshop on Computer Algebra in Scientific Computing, Konstanz*, pages 213–232. Springer-Verlag, July 2001.
9. Manfred Göbel, Wolfgang Küchlin, Stefan Müller, and Andreas Weber. Extending a Java based framework for scientific software-components. In V.G. Ganzha, E.W. Mayr, and E.V. Vorozhtsov, editors, *omputer Algebra in Scientific Computing, CASC'99. Proceedings of the Second International Workshop on Computer Algebra in Scientific Computing, Munich*, pages 207–222. Springer-Verlag, May 1999.
10. Roman Maeder. *The Mathematica Programmer*. AP Professional, 1994.
11. Modelica Association. Modelica – a unified object oriented language for physical systems modeling. language specification version 2.0. Modelica Association, http://www.modelica.org/, February 2002.
12. The Common Object Request Broker: Architecture and specifikation. Technical report, Object Management Group (http://www.omg.org/), 1998.
13. Andreas Weber, Gabor Simon, Wolfgang Küchlin, and Jörg Hoss. Lessons learned from using CORBA for components in scientific computing. In Ganzha et al. [7], pages 409–422.
14. Daoqi Yang. *C++ and Object Oriented Numeric Computing for Scientists and Engineers*. Springer-Verlag, 2000.