Global J. of Engng. Educ., Vol.9, No.1
Published in Australia

Using the PhysBeans Toolkit to Construct Applets for Physics Simulation*

Peter Junglas
Private Fachhochschule für Wirtschaft und Technik Vechta/Diepholz - Private University of Applied Sciences
Schlesierstraße 13a, D-49356 Diepholz, Germany

Constructing a simulation program that can be used for teaching physical phenomena entails a lot of work. The concepts of graphical programming, as known from JavaBeans or Visual Basic, could help in this instance, as they can reduce the programming task mainly to selecting a number of predefined blocks and connecting them with the mouse. PhysBeans is a library of such building blocks specially designed for the simple construction of physical simulations. It is based on the Java Beans model and contains blocks for standard input and display elements, for physical models and their visualisation and for mathematical functions. The creation of an example applet for the simulation of an electric dipole is demonstrated in this article so as to show the benefits of this approach. The starting point is a graphical block diagram that shows the used building blocks and the flow of information between them. This diagram can simply be rebuilt in a graphical programming environment, reducing the amount of handwritten source code significantly. PhysBeans has been used to create applets for many different physical areas. Notably, the library and all applets have been released as open source.

INTRODUCTION

Physical simulations using Java applets can be a useful didactic tool that has already been proven by many applications (eg [1][2]). Unfortunately, the construction of such applets is a huge programming effort and needs a profound knowledge of the Java programming language and its libraries. For this reason, large lists of applets have been collected on the Internet; these can be used freely (eg [3]). However, every lecturer has his/her own way of explaining things and different points to make, which lead to a large number of similar applets that have all been created from scratch.

A way out of this dilemma are physlets, which are a set of general purpose applets that each span a certain physics topic [4]. They can be configured using JavaScript to such a wide degree that very different-looking applications can be constructed from one basic applet. But this freedom has its price: in addition to JavaScript, one has to learn the special features of the specific physlet.

The approach discussed in this article has a different focus: The PhysBeans library contains a set of building blocks that allow a physical simulation to be constructed in a graphical way. Little (hopefully none) explicit Java code is needed to create an individual applet that exactly suits the needs of a lecturer. A complete examplary applet will demonstrate what the PhysBeans library provides and how it is used.

ELECTRICDIPOLE: AN EXAMPLE APPLET

The applet that serves as an example is used in a basic course on electromagnetic fields for engineers. It shows the electric field of two opposite point charges. The electric field is displayed by a grid of arrows, its absolute value is represented using a colour table. The value of the charges and their separation can be changed with sliders or by direct numerical input. This gives a clear intuitive understanding of the qualitative properties of the dipole field (see Figure 1).

*A revised and expanded version of a paper presented at the *8th Baltic Region Seminar on Engineering Education*, held in Kaunas, Lithuania, from 2 to 4 September 2004. This paper was awarded the UICEE gold award (joint third grade with two other papers) by popular vote of Seminar participants for the most significant contribution to the field of engineering education.

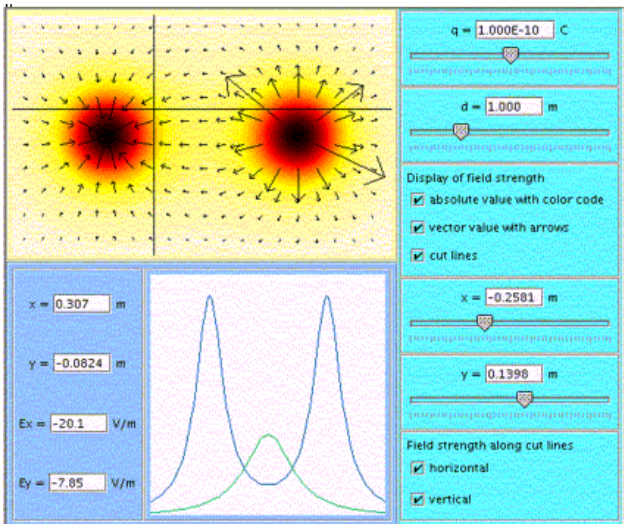

Figure 1: An applet of an electric dipole.

Next, students have to identify the quantitave behaviour in the far and near field regimes. For this purpose, the applet allows to measure the electric field strength at an arbitrary point by simply clicking there. Furthermore, the absolute value of the field strength along arbitrary horizontal or vertical cut lines is displayed as simple graphs. This permits the use of the applet as a *virtual experiment*, as referred to in refs [5][2].

The programming of such an applet entails a great deal of work; altogether it uses 47 classes (plus many Java standard classes) consisting of 5,049 (non-empty) lines of code, only a minor part of which (13%) is connected with the physical equations or basic mathematics. The rest is mainly used for the user interface and the graphical representation of the data and for general infrastructure purposes such as coordinate transformations, colour tables or message passing facilities. Almost all of the classes can be (and actually are) reused in other applets. The applet specific part (the *main program*) only makes up 6% of the code. This clearly shows the large potential for a classical class library to considerably simplify the task of an applet programmer [6]. Yet this can be made even simpler by utilising a graphical approach.

GRAPHICAL PROGRAMMING WITH JAVABEANS

Programming infrastructures like Visual Basic or JavaBeans try to reduce the programming task as much as possible to simple graphical operations [7][8]. They are especially helpful for the implementation of graphical user interfaces, but can also be used in a much broader sense. For instance, writing a program in a JavaBeans environment is reduced mainly to the following steps:

- Select the needed components (the Java beans proper) from a palette of predefined beans;
- Configure the beans by entering all non-default values for their properties in a property sheet;
- Connect the beans with lines that denote messages going from one bean to another, and define the message content.

Beans have a built-in mechanism to send messages to a list of receivers whenever their state changes or an external (eg user-initiated) event occurs. Different programming environments provide various means allow messages to be defined without writing any explicit code. Nevertheless, it is often necessary, and sometimes even simpler, to write some Java lines by hand. The following example illustrates how to work with JavaBeans using the free Netbeans environment [9]. It shows clearly what can be done graphically and where hand code still is necessary (see Figure 2).

The example program is a simple calculator with two input fields to enter numbers, four buttons with the basic arithmetic operations and an output field displaying the result. The construction of the program proceeds with the following steps: start by using a simple template for applets. An empty grey rectangle is shown to represent the graphical user interface.

Next, add the necessary components from the palette by selecting them and clicking in the rectangle, namely three text fields, three labels, four buttons and five panels, simple containers used for nice grouping of the elements.

The components can now be configured by providing values for all properties that are different from their defaults. This is carried out by simply editing the values in the property sheets. The changed properties cover the following:

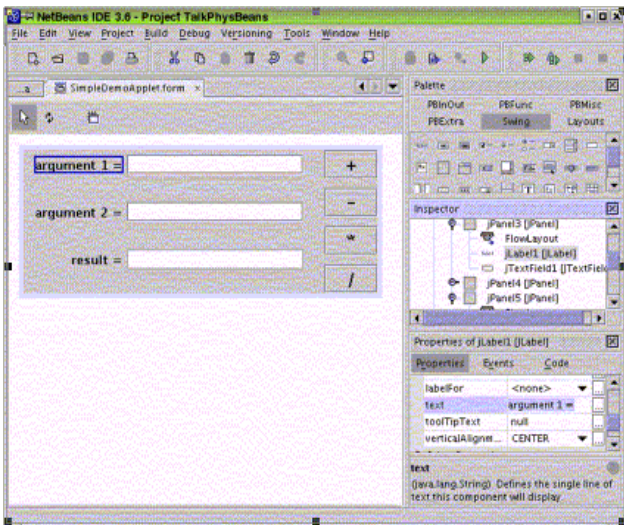

Figure 2: Programming environment NetBeans.

- Layout types, border sizes, etc, to arrange everything nicely;
- Text and font of the labels;
- Size, font and initial text (none) of the text fields;
- Text and font of the buttons.

This completes the user interface displayed in a WYSIWYG style in the NetBeans window.

Finally, behaviour is added by drawing connections and defining the proper messages. Four connections are needed here, one from each button to the output field. The message should say basically: *When I* [the button] *am pressed, you* [the output field] *have to get the numbers from the two input fields, add* [subtract, etc] *them and display the result.*

This message is a bit too complicated to create it graphically, although this would be possible by inserting an intermediate bean for addition. To see what is possible without any hand-coding, the following simpler message is inserted: *When I* [the button] *am pressed, you* [the output field] *have to get the number from the first input field and display it.*

After clicking the sender (button) and receiver (output field) with the connection tool, the *connection wizard* pops up and asks for the following information, which can be entered simply by selecting from a list of predefined values, namely:

- Reason of the message (button is pressed);
- Property to change at the receiver (text);
- Origin of text (text property of first text field).

This creates all the code needed for the message transfer as guarded code (ie it cannot be changed manually) and the message content itself:

```
jTextField3.setText(jTextField1.getText());
```

Now the original message can be implemented easily by replacing the code with the following lines:

```
double val1 = Double.parseDouble(jTextField1.getText());
double val2 = Double.parseDouble(jTextField2.getText());
double result = val1 + val2;
jTextField3.setText(result + "");
```

The applet is finished by proceeding in the same way with the other three connections. The complete program consists of 139 non-empty lines, of which 16 have been hand-coded.

COMPONENTS OF THE PHYSBEANS LIBRARY

The use of this graphical programming style is only possible if all the necessary beans exist. Standard environments usually contain beans from Sun's Swing library, which serve as basic building blocks for a graphical user interface [10]. To construct physical simulations, one needs more sophisticated input and output elements, plus many non-graphical components. This is where the PhysBeans library comes in.

The aim of the PhysBeans project is to provide a set of beans that make possible a graphical construction of physical simulation applets. Its components fall into five categories, namely:

- Input beans allow the input of information by the user. They are displayed graphically, usually in a special input panel, eg TextSlider, Timer.
- Output beans display the results of measurements or show views of physical objects, eg Oscilloscope, VectorTextField, LayeredScreen.
- Physical beans describe physical objects and their behaviour using specific physical laws. They do not contain any graphical representations, eg ElectrostaticPointCharges, MathPendulum.
- *View beans* define a specific graphical representation. Often they are the interface between a physical object and an output element, usually the LayeredScreen, eg ImagePainter, VectorPainter, FunctionPainter.
- Function beans represent mathematical functions. They usually have one or more inputs (arguments) and an output sending the result. They are not displayed graphically, eg FFTFunction, CutFunction.

The task of arranging all elements efficiently needs a good understanding of the Java layout mechanism. So as to free the programmer from this burden, the library additionally contains three applet templates with different predefined arrangements of panels for input and output elements and for the visualisation of the physical objects.

The beans that are described in more detail below offer a good overview of the kind of components that the PhysBeans library provides. Furthermore, they are the building blocks of the two applets described.

Input beans include the following:

- *TextSlider* allows the entry of a numerical value using a slider or a text field. Some of its properties are a description and a unit text, the number of significant digits to use for the value and the type of its slider (logarithmic, inverted). It sends a message when a new value is entered.
- *SwitchBox* is a collection of options that can be selected independently. It facilitates a nice arrangement and sends a message whenever a box is clicked.

- *Timer* is a bean that sends regular events to drive dynamical simulations or animations. Its user interface allows the animation speed to be changed, paused or continued, as well for the animation to be restarted.

Output beans cover the following:

- *VectorTextField* displays a complete vector of numerical values, each in its own text field. It provides description and unit texts and some layout options.
- *LayeredScreen* is a screen with several layers, each of which is attached to a different view. Views are drawn in fixed order and can be disabled individually. The LayeredScreen defines a common world *coordinate* system for all its views, cares for the transformation between screen and world coordinates and has an option to display a crosshair cursor. It listens to mouse clicks and sends messages with the corresponding world coordinates.
- *Oscilloscope* shows a number of curves as animated functions of time. It allows the curves to be scaled and to make measurements by clicking on the oscilloscope screen.

Physical beans incorporate the following:

- *ElectrostaticPointCharges* is a model of a set of static electric point charges in two dimensions. It computes the electric field strength as a vector field and the absolute value and the potential as scalar fields. The number of charges, their value and their positions, can be configured. For convenience, it provides common scale factors for charges and position vectors. It sends a message to notify of general changes and can be asked to send a message with the field strength at a given point.
- *MathPendulum* describes the motion of a mathematical pendulum by solving its differential equation. The mass, length, gravity acceleration and friction coefficient can be set. Also, it can be connected to an external function that provides arbitrary time-dependent values for a driving torque.

View beans involve the following:

- *ImagePainter* draws a colour image that represents a scalar field by using a colour table to map scalar values to colours. The standard map defines a cold-hot feeling going from blue over white to red.
- *VectorPainter* draws arrows at regular grid points representing a vector field. If the length of an arrow is larger than a given limit, the arrow is hidden or a marker may be drawn instead.
- *FunctionPainter* draws a vector of two-dimensional points by connecting them with lines of a given colour. It is commonly used to create function graphs.
- *SimpleFigurePainter* adds simple additional features to an image like a few lines, a rectangle, an oval or an arc (filled or unfilled), or a string. It is a graphical *Swiss army knife* for all the little things to add. Its properties contain the type of figure, the points defining it and its colour.
- *PendulumView* displays a pendulum as a circle attached to a line. An option is the drawing of an arrow of varying length connected to the circle, thereby representing an external force.

Function beans include the following:

- *GainFunction* multiplies its input with a fixed number.
- *BooleanSelector* creates an output vector of logical variables by selecting some of its input values and reordering them.
- *ReduceAngleFunction* subtracts multiples of 2π from the input value, until the result lies in the basic interval $[-\pi,\pi]$. The function can be switched off, upon which it simply passes its input value to the output.
- *CutFunction* is a specialised function that computes a number of field values of a given scalar field along a horizontal or vertical cut line. It outputs a vector of two-dimensional points, each giving the coordinate along the line and the corresponding field value. The position and orientation of the line can be configured, as well as the number of points to compute.

CONSTRUCTION OF AN APPLET

The development of a new applet (eg electric dipole) starts with the following preliminary steps:

1. Didactical considerations: What are the basic ideas to learn from it? How can it be used?
2. Design of the user interface: What quantities can be entered, what measurements can be done, what is displayed? This fixes the input and output beans and most of the view beans.
3. Internal function: Which beans are needed to make everything work? Starting with the central physical

beans, all necessary mathematical and auxiliary function beans are added.

At the end of this stage, a flow chart is constructed that shows all necessary beans and the message flow between them (see Figure 3). The upper part shows the ElectrostaticPointCharges bean, which defines the two opposite charges, whose value and distance can be set using TextSliders. The ImagePainter and VectorPainter create the corresponding colour and arrow images that are displayed on the upper LayeredScreen. Additional elements here are the two perpendicular cut lines, which are created with a SimpleFigurePainter and whose positions are set with TextSliders. A SwitchBox can be used to choose what will be displayed on the screen.

The lower section of the flow chart contains two CutFunctions that compute the values along the given cut lines. They send their results to FunctionPainters, which create the graphs that are displayed on the second LayeredScreen. The values are recomputed every time the x- and y-sliders send new values, or when the physical model is changed. Again, a SwitchBox is utilised to choose which graph to show.

If the user clicks on the upper LayeredScreen, the coordinates are displayed by a VectorTextField. Furthermore, they are forwarded to the ElectrostaticPointCharges bean, which in turn sends the corresponding field values to another VectorTextField.

The implementation now proceeds in the same way as in the calculator example:

- Start with a standard template.
- Add all visible and invisible components by selecting and clicking on the proper panel (these can easily be rearranged later with drag and drop).
- Configure all the beans by defining, for example, the two charges with the opposite values, all the description texts and the ranges of sliders and the coordinates of the screens.
- Define messages for all connections in the flow chart.

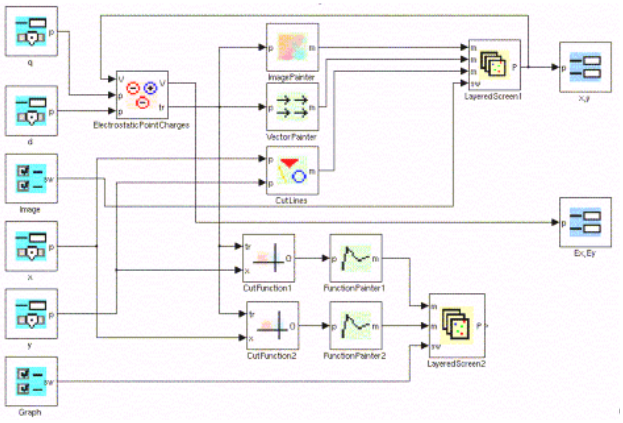

Figure 3: Flow chart of an electric dipole applet.

As mentioned above, the last step is the only one that needs any explicit Java code. For example, the message from the TextSlider *y* to the SimpleFigurePainter *CutLines* cannot be done in a graphical way directly, because the slider sends a simple number (a double), while the SimpleFigurePainter needs two points to define the vertical line. Such a conflict can be resolved manually with the following lines:

```
double val = textSlider3.getValue();
simpleFigurePainter1.setP3(new DVector(-1.0, val));
simpleFigurePainter1.setP4(new DVector(1.0, val));
```

If one wants to avoid any hand-coding completely, a simple function bean could be used as an interface, which creates a two-dimensional vector from a number by getting the second coordinate as a fixed parameter.

The complete applet program consists of 277 non-empty lines, of which 21 lines are used to define message contents. Out of these, only seven lines are created manually, but this could be made to zero with little additional effort.

MATHPENDULUM: A DYNAMICAL APPLET

The second applet example simulates the motion of a swinging pendulum and is used in a course on nonlinear and chaotic systems. Since its flow chart is simpler, it is better suited to show explicitly how one can handle the interface problems discussed above. However, its simplicity is superficial; the true complexity needed for the differential equation solver is hidden in the MathPendulum bean.

The applet shows directly the motion of the pendulum as a simple animation, the angle and angular velocity are displayed as functions of time on an oscilloscope (see Figure 4).

The user of the applet can pause, continue or reset the timer and change the animation speed over a wide range. The user can also set the initial values and the physical parameter g/l, as well as utilise the oscilloscope measurements by clicking on a curve. One can also select the curves that should be displayed and scale them to adapt to different ranges of the functions. In addition, the angle can be reduced to the interval $[-\pi,\pi]$, which leads to a better representation

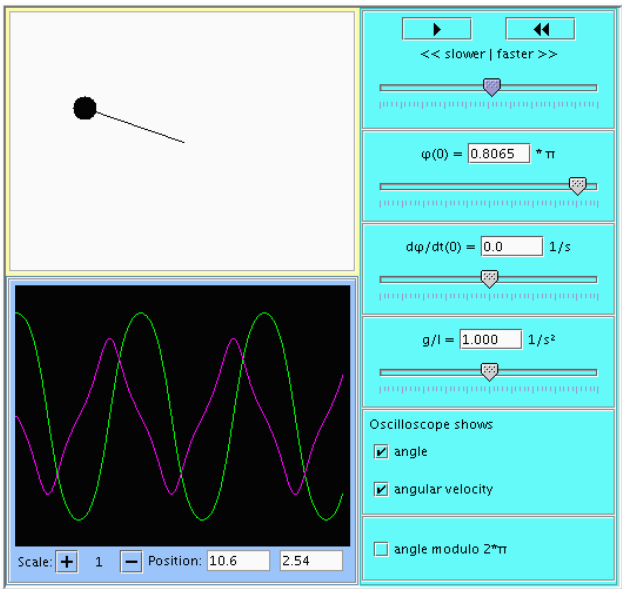

Figure 4: An applet *mathematical pendulum.*

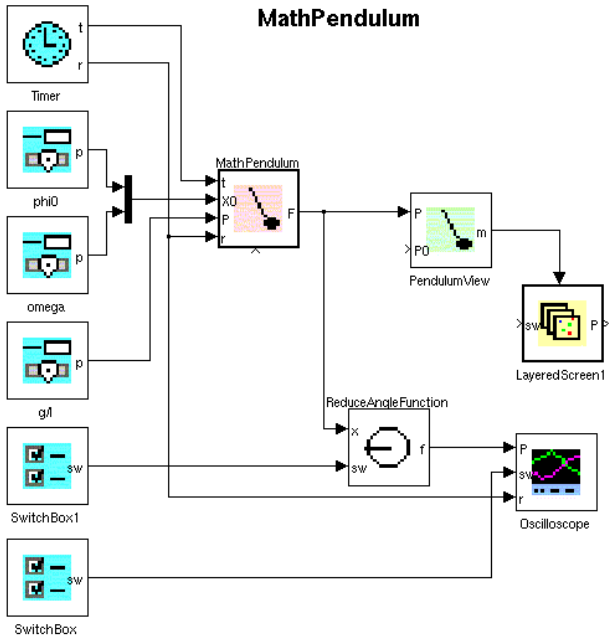

Figure 5: Flow chart of the mathematical pendulum applet.

of an overswinging motion.

Most of the necessary beans for this applet correspond directly to the components of the user interface. To make everything work, three more beans have to be added, namely:

- The MathPendulum bean is the central bean as it contains all the information about the physical system, such as the mass and length of the pendulum, as well as its time behaviour.
- The PendulumView contains the *optical* information of the system. Given the angle, it draws a simple pendulum.
- The ReduceAngleFunction optionally reduces large angles to a fundamental period.

The flow chart in Figure 5 displays all the necessary beans and their connections.

The Timer periodically sends messages with the next time value to the MathPendulum bean. This computes the corresponding new state (angle φ and angular velocity ω) and sends it to the PendulumView and the ReducedAngleFunction. The PendulumView draws an animated image of the pendulum using the LayeredScreen. The ReducedAngleFunction passes its values to the Oscilloscope, which may be changed according to the state of the SwitchBox1.

This simple description conceals some problems that are typical for many applets, namely:

- The range of the TextSlider setting the initial angle is [-1,1]; its unit (which is just a text string) is π. This makes the entered number more intuitive, but creates an interface problem between the TextSlider output and the MathPendulum input.
- The ReduceAngleFunction operates on a vector, which contains two values: (φ, ω). Therefore, its internal switch variable also is a two-dimensional vector. However, the SwitchBox only has one option for the angle as its output vector is one-dimensional. The other switch should, of course, always be set to *false*, since it is meaningless to reduce the value of the angular velocity.
- The output of MathPendulum is 2d: (φ, ω). The input of PendulumView is 2d as well, but it is (φ, M), where M is an optional external torque, which is represented by an arrow.

One can easily cope with these interface problems by manually modifying the message code, eg by using bean methods that give direct access to its variables individually or by explicitly creating the necessary vectors. The problems in the example can thus be solved with the following lines:

```
1. mathPendulum1.setPhi0(textSlider1.getValue()*Math.PI);
2. reduceAngleFunction1.setProcessed(0,switchBox2.
   getSelected(0));
3. pendulumView1.setAngle((mathPendulum1.
   getOutputValues(0));
```

If one wants to adhere to graphical methods, then one has to include the following interface beans:

- The necessary multiplication with π can be achieved by the GainFunction bean, which simply multiplies its input with a fixed number.
- The BooleanSelector bean allows the input and output vector dimensions to be defined differently. Any input variable can be routed to one (or more) output variables; the remaining output switches can be set to a constant value.
- One could add a similar Selector bean for the third problem as well, but a special feature of the PendulumView makes it possible to use the following trick: since the vectors have the same size, one can directly connect MathPendulum and PendulumView. One can then set the *showArrow* property of PendulumView to *false* so that the second input is simply ignored.

These additions are represented in the final flow chart (see Figure 6).

CONCLUSIONS

The above examples show that the concepts of JavaBeans, together with a proper set of components like PhysBeans, make it possible to create non-trivial applets of high didactic value, almost without any explicit programming.

The notion of reusable blocks is especially suited for general controls and displays but can also be applied to create all necessary internal parts of a virtual experiment. To dispense from the need of writing Java code at all, one has to further address the following points:

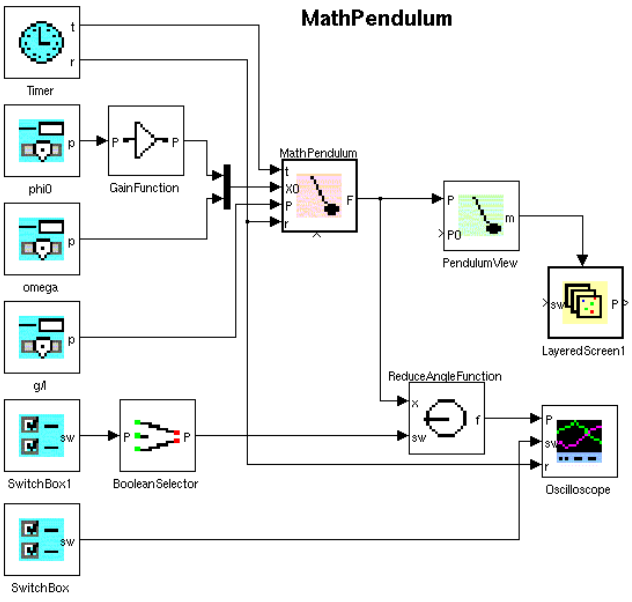

Figure 6: Final flow chart of the mathematical pendulum applet.

- The need of interfacing between different blocks can be reduced to a minimum by the overall design of the kind of messages and by adding a few simple arithmetic blocks.
- The number of custom blocks needed in special situations can be reduced by providing general-purpose blocks, which can be configured widely.
- Some of the messages could be easily created graphically with better support from the programming environment. An open source program, such as NetBeans, is probably a good starting point for the necessary extensions.

The PhysBeans library has been used for the applets on the supplemental CD-ROM of ref. [11] and for a course on non-linear and chaotic systems. It is the basis of a forthcoming multimedia-enhanced course on vibration theory. Existing applets are mainly from linear and non-linear oscillations, but applets for electrodynamics, optics, thermodynamics and nuclear physics prove its wide applicability. They all can be downloaded from the author's PhysBeans homepage [12].

Future developments will concentrate mainly on two points, namely: building of new beans (eg with 3d graphics) and new applets; and the redesign of the underlying code to make use of the recent OpenSourcePhysics library, which contains many tutorials that help to create new beans [6].

The PhysBeans project is still at an early stage. The library and all the applets have been released as open source so as to make it more useful and to proceed faster. It could be a starting point to make physics educators, who want to use applets, more productive by largely simplifying the programming tasks, so that one can concentrate on the design of didactically useful and easily adaptable physical simulation programs.

REFERENCES

1. Department of Didactics, University of Erlangen, Applets: Vorstellung, http://www.didaktik.physik.uni-erlangen.de/download/applets.htm
2. Junglas, P., Using applets for physics education: a case study of a course in non-linear systems and chaos. *Proc. 7th Baltic Region Seminar on Engng. Educ.*, St Petersburg, Russia, 61-64 (2003).
3. Physics Web, Interactive Experiments,

http://physicsweb.org/resources/Education/Interactive_experiments/
4. Christian, W. and Belloni, M., *Physlets - Teaching Physics with Interactive Curricular Material*. Upper Saddle River: Prentice Hall (2001).
5. Junglas, P., Einsatz von Applets in der Physik-Ausbildung – Fallstudie *Nichtlineare Systeme und Chaos*. *Global J. of Engng. Educ.*, 7, **3**, 337-347 (2003).
6. Open Source Physics, http://www.opensourcephysics.org/
7. *Microsoft Visual Basic 6.0 Reference Library*. Redmond: Microsoft Press (1998).
8. Englander, R., *Developing Java Beans*. Farnham: O'Reilly (2002).
9. Boudreau, T., Glick, J. and Greene, S., *NetBeans*. Sevastopol: O'Reilly & Assoc. (2002).
10. Walrath, K., Campione, M. and Huml, A., *The JFC Swing Tutorial*. Boston: Addison-Wesley Professional (2004).
11. Stöcker, H., *Taschenbuch der Physik mit CD-ROM*. Frankfurt am Main: Harri Deutsch (2004).
12. Junglas, P., PhysBeans Homepage, http://www.peter-junglas.de/fh/physbeans/index.html

BIOGRAPHY

Peter Junglas was born in 1959. He studied physics in Hannover and Hamburg and specialised in mathematical physics. In 1987, he obtained his PhD with Prof. Buchholz at the University of Hamburg with a topic that covered general quantum field theory.

After spending time at the University of Goettingen and the MPI for Aeronomy in Katlenburg/Lindau, he worked at the Computing Centre of the TU Hamburg Harburg until 2000. The emphasis of his activities while there incorporated in the fields of scientific computing and parallel programming.

Since 2000, he has been Professor of physics and computer science at the Department of Mechanical Engineering at the Fachhochschule für Wirtschaft und Technik (FHWT), which is a private university of applied sciences in Vechta/Diepholz/Oldenburg.

His present interests cover the development of multimedia techniques for teaching, as well as the broad use of simulation techniques.