DeApp - An Application in Java for the Usage of Differential Evolution

by Rainer Storn[1)]

Abstract

This document contains a brief overview of the Java based application DeApp. The latter is set out to provide an easily extendable and platform-independent framework to solve function optimization problems with Differential Evolution (DE).

[1)]International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704-1198, Suite 600, Fax: 510-643-7684. E-mail: storn@icsi.berkeley.edu.

1.1. Introduction

This document covers the design of an application for the usage of Differential Evolution (DE) in Java®, called DeApp. The objective of DeApp is to provide a framework upon which a user can program his own cost functions, minimize them with DE and view the parameters, cost function or other valuable optimization information in a graphical way, if this is wanted. DeApp comes with a graphical user interface which makes it convenient to change DE's control variables and select the various ways to monitor the optimization. An overview of the current version of DeApp (1.0), as it appears on the desktop when launched, is shown in Figure 2-1.

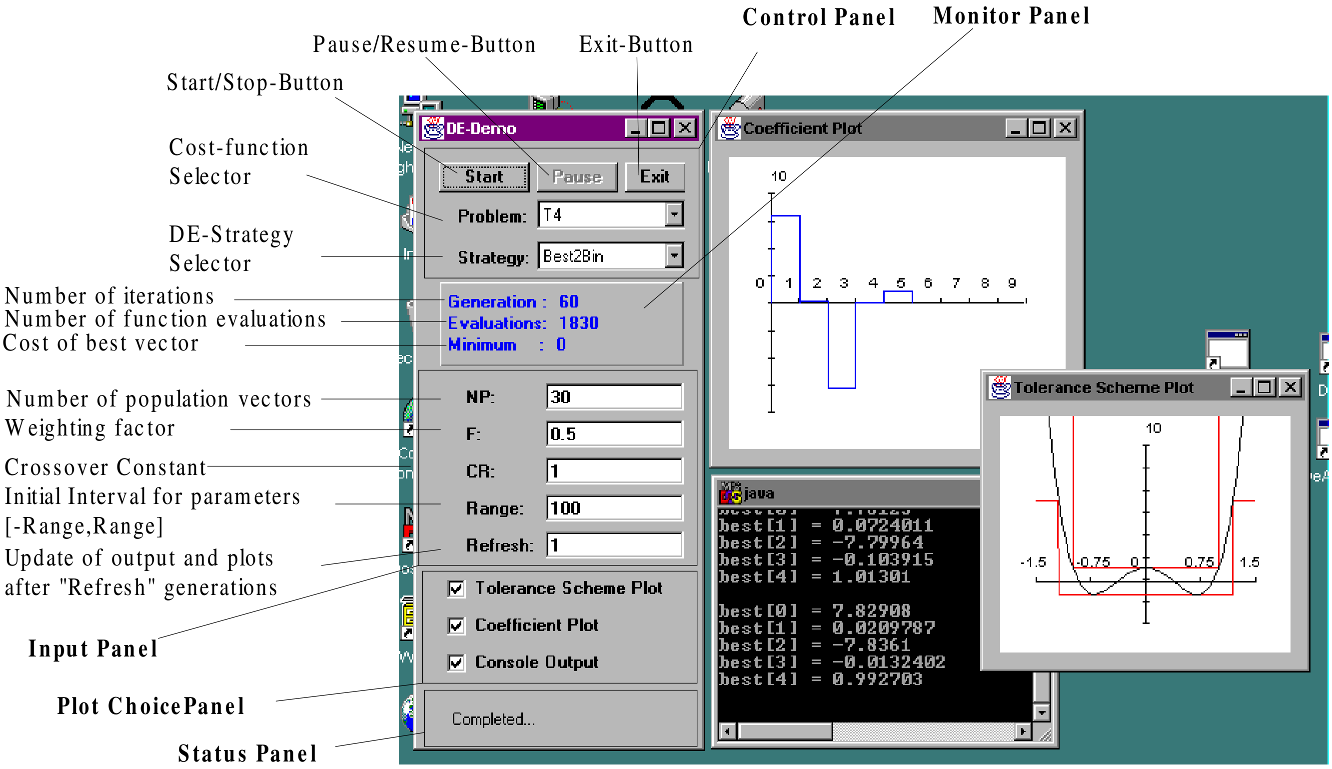

Figure 2-1: Appearance of DeApp1.0 on the desktop.

DeApp has been written for the public domain, so the executables as well as the source code are publicly available. Modifications of the program are not only allowed but also encouraged. It is hoped that public availability will stimulate and foster the development of future versions of DeApp.

1.2. How to run the optimization

The optimization is prepared by selecting a problem, i.e. cost function from the cost function selector and a DE-Strategy from the DE-Strategy selector. Then the number of poulation vectors NP, the weighting factor F and the crossover constant CR must be chosen. This is done by simply typing the desired values into the pertinent text fields.

As a rule of thumb NP is chosen to be 5 to 10 times the number of parameters, but sometimes even higher values are required. F is dependent on the DE strategy chosen, but values smaller than 0.5 almost never occur. For the strategy DE/Best/2/bin F=0.5 is usually a good choice. For DE/Rand/1/bin values F=0.7 or F=0.8 are often more productive. Again, as a rule of thumb, if NP is increased strongly then F should be slighlty reduced. The crossover constant CR stems from the range [0,1] but is usually chosen to be either large, i.e. 1.0 or small, e.g. 0.2.

The variable “Range” defines the interval from which the parameter vectors are initialized. If Range=100, then the parameters are drawn randomly from the interval [-100, +100]. The variable “Refresh” denotes the monitor panel update frequency as well as the plot screen update frequency. If Refresh=1, then the monitor panel as well as the plotting screens are updated after every iteration (generation) of the optimization. If Refresh=10 then the update occurs every 10 iterations. The plotting screens as well as the console output can be activated and deactivated during the optimization by clicking on the appropriate check boxes of the PlotChoicePanel. As the optimization is running the status panel shows the string “Running”. When the optimization is complete the string “Completed” shows and the parameters of the best parameter vector are written into a file named “DEOut.dat”.

1.3. Drawing Symbols

To make it easier for a user to make changes to DeApp according to his/her needs more information than just the source code is provided. The drawing symbols to explain DeApp's structure used in that document are summarized in Figure 2-2.

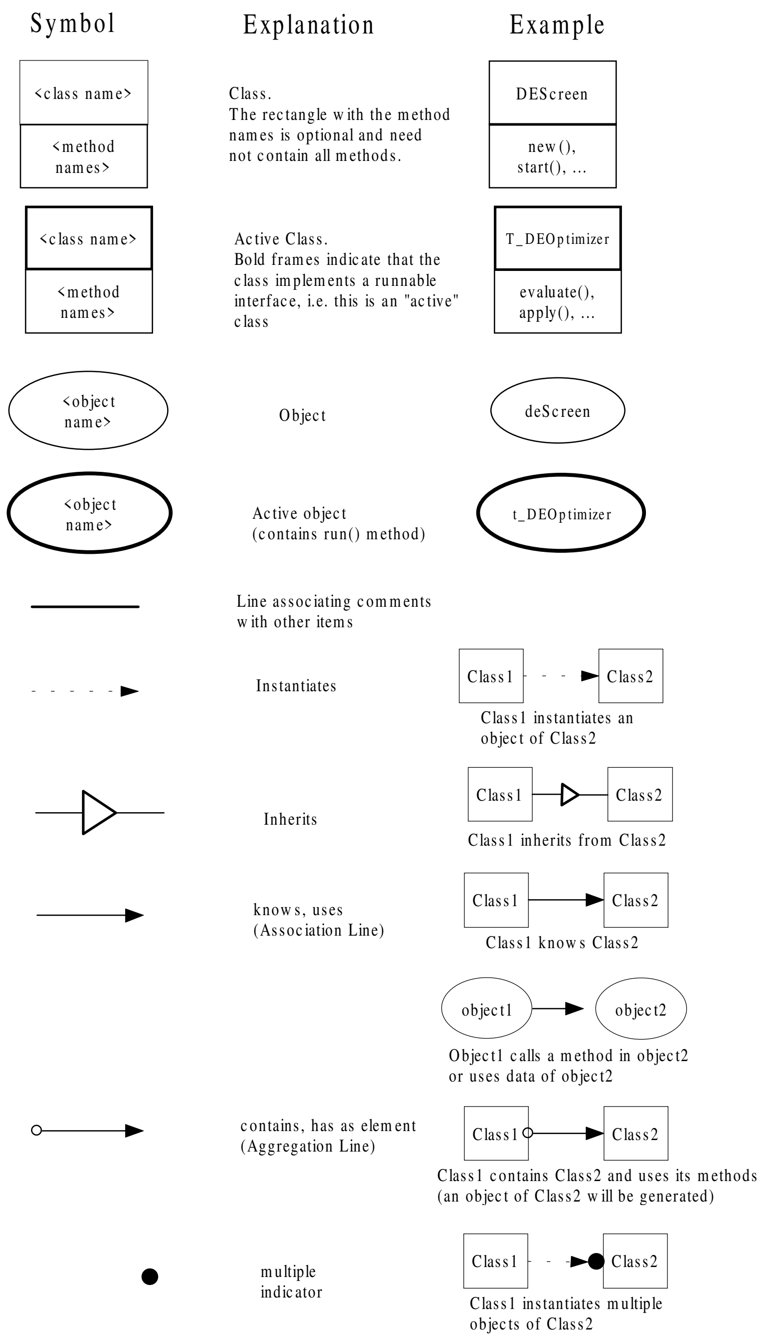

Figure 2-2: General class- and object-related drawing symbols.

1.4. Class Structure of DeApp

Figure 2-3 shows the class structure of DeApp. The central component is DEScreen which acts as a mediator. The various objects always communicate via DEScreen which makes them loosely coupled.
Also the plotting has been decoupled entirely from the optimization. Although this has the disadvantage that some functions might have to be implemented more than once, it has the advantage that plotting can be added independently of the actual optimization. This is very helpful especially for new optimization problems where plotting might be added later, or even be omitted completely for the sake of optimization speed. DeApp also offers the possibility to add as many plotting screens as desired. To attain flexibility the plotting screens are all independent and not coupled by inheritance. This makes it possible to use different plotting programs for the different screens.

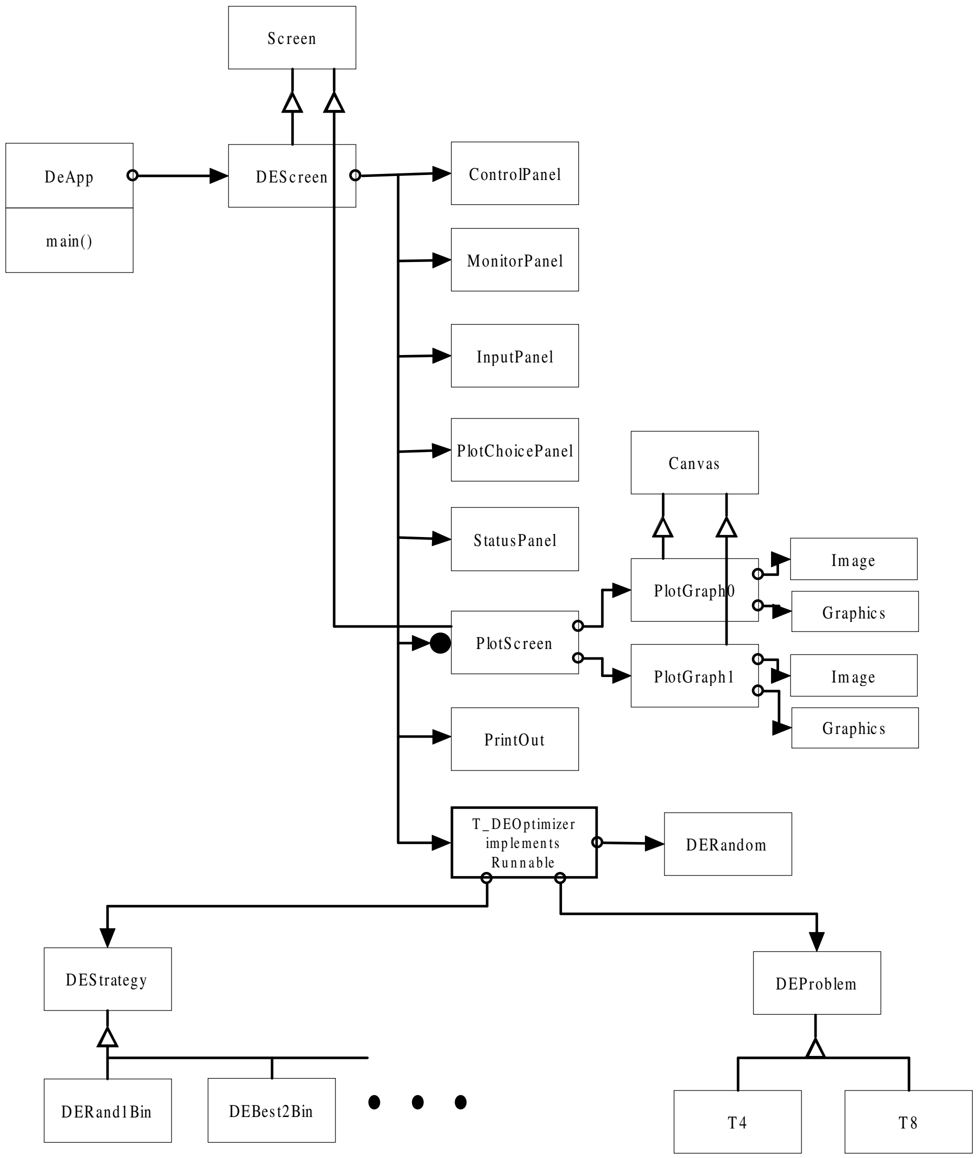

Figure 2-3: Class structure of DeApp.

1.5. Sequence chart for a simple optimization scenario

Figure 2-4 depicts a simplified sequence chart which shows how the various objects in DeApp interact by calling their methods. At first the main() method in DeApp generates the object “deScreen” which is the hub of the entire program. Object “deScreen” instantiates all the panels like “controlPanel”, “monitorPanel”, “inputPanel”, “plotChoicePanel” and “statusPanel”. It also instantiates the object “printOut” which serves to print the final parameter values of the best parameter vector into a file named “DEOut.dat”. Finally the active object “t_DEOptimizer” which runs the actual optimization thread is generated. “t_DEOptimizer” itself generates three additional objects, “deRandom”, the random number generator, “Strategem” which defines the DE-Strategy selected in the control panel, and “deProblem” which selects the cost function that has to be minimized.

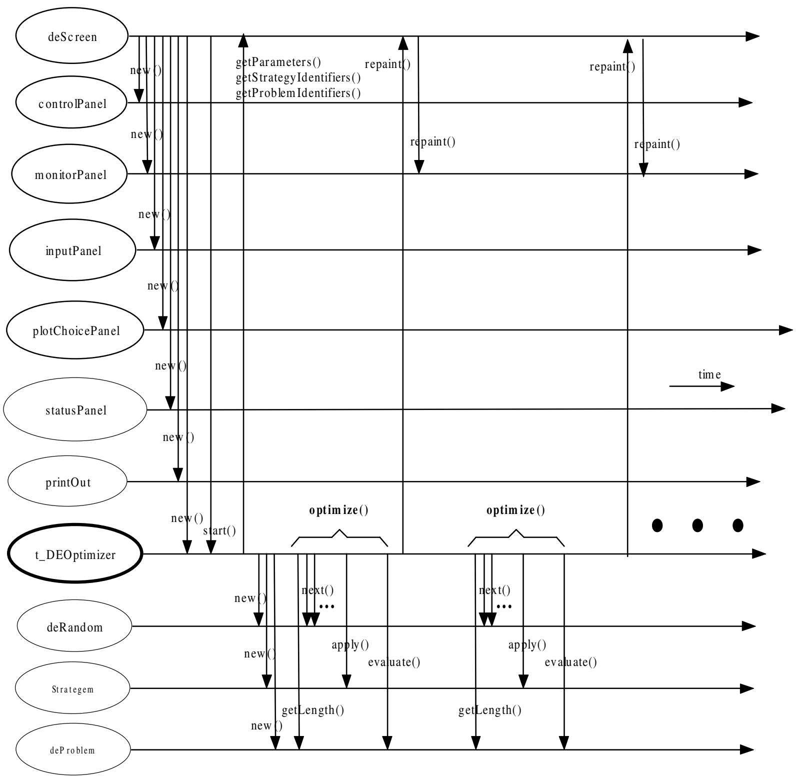

Figure 2-4: Simplified sequence chart for the operation of DeApp.

The central method in “t_DEOptimizer” is optimize() which does the actual optimization. After a certain number of generations, which is defined by the variable “Refresh”, the monitor panel is updated via the repaint() method.
Note that in this example scenario the plotting functionality has been completely neglected. Indeed it is not necessary for DeApp to include the plotting, the optimizer can be operated completely without plotting.

1.6. Naming conventions in the source code

In order to make the source code easier to understand and maintain, several naming conventions have been adhered to. Table 1.6-1 lists the current naming conventions. In a few instances the naming conventions have not been followed due to historical reasons and for the sake of a speedy delivery of DeApp. In future releases of the software also these deviations will be adapted to the naming conventions.

Item	Naming Convention	Example
Classes	Start with upper case. Each new concatenated word starts upper case.	MyNewClass
Active Classes	Like other classes but name contains T_ somewhere to indicate the thread behaviour.	T_MyActiveClass
Objects	Start lowercase. Each new concatenated word starts upper case.	instanceOfClass
methods	Start lower case. Each new concatenated word starts upper case.	getTimeOfDay()
variables	All lower case. Words are concatenated with underscore.	my_printer
final variables	all uppercase	MAX_NUMBER

Table 1.6-1: Naming conventions for the DeApp development.

1.7. Altering the program to run a new problem

Modifying the program to minimize a new cost function is fairly simple and straightforward. Let's assume that we want to get rid of the problems T4 and T8 which are the example problems of the current release 1.0 of DeApp. Simply remove these classes from the code and insert another class, e.g. MyProblem, which is programmed along the lines of T4 or T8. This means the following:

1. The number of parameters "dim" has to be defined in the constructor of MyProblem.
2. The method

   ```
   public boolean completed()
   ```

   contains the stopping criterion. A common way to do this is to define a value to reach (called "mincost" in the code) and return TRUE if mincost is smaller than the value to reach.
3. The method

   ```
   public double evaluate (double temp[], int dim)
   ```

   contains the actual cost function where temp[] is passed a parameter vector with dim elements. evaluate() returns the cost function value.
4. In the class DEScreen the variable

```
public String problem_identifier[]   =
{
     "T4",
     "T8"
```

```
};
```

which steers the content of the cost function selector in the control panel has to be adapted too. In our simple example it would have to be changed to

```
public String problem_identifier[]   =
{
     "MyProblem"
};
```

The plotting functions set up for T4 and T8 are have, of course, to be adapted to the new problem. Hints for doing this will be provided below. The console output is always valid, independent of the problem because it merely outputs the best-so-far parameter vector.

1.8. Making changes in the optimizer

Sometimes one may want to change certain things in the optimizer itself, e.g. currently there are maximum and/or minimum values defined for NP, F, CR, Range, and dim (among other variables). There are two locations where these values have to be taken care of (which is not optimal and will be changed in the future). One is in the class InputPanel where the following holds:

Variable	Minimum value	Maximum value
NP	0	NPMAX=200
F	0	FMAX=1.0
Cr	0	CRMAX=1.0
Range	0	RMAX=500.0

The other location is in the class T_DEOptimizer where we currently have

Variable	Maximum value
NP	MaxN=300
dim	MaxD=17

The maximum values of all the variables except CR can be increased as needed although an increase for FMAX probably doesn’t make sense.

Another interesting location for a change is the code section

```
try
 {
   action.sleep (DEProblem.NAPTIME);
 }
 catch (Exception E)
 {
    System.err.println (E);
 };
```

in the run() method of T_DEOptimizer. This code section is required only to allow for a reasonable pause between graphics updates (NAPTIME is set to 10ms in the class DEProblem). If graphics shall not be used, it is advisable to comment out this section which results in a speed increase of the program. An even simpler way to increase the speed of the optimization is to increase the value for the Refresh constant. When doing this, less processing power is devoted to the output of intermediate results.

If a new DE-Strategy shall be added a new class must be written which extends the class DEStrategy. If one of the current strategy classes like DEBest2Bin is looked at it becomes obvious how to do the change. If the new strategy, let's call it MyStrategy, is added also the variable

```
public String strategy_identifier[] =
  {
    "Best2Bin", "Rand1Bin", "RandToBest1Bin"
  };
```

in class DEScreen has to be changed to

```
public String strategy_identifier[] =
  {
    "Best2Bin", "Rand1Bin", "RandToBest1Bin", "MyStrategy"
  };
```

1.9. Adapting the graphics

The graphics part for DeApp is kept pretty simple. There are more sophisticated graphics packages around, e.g. Ptplot1.2p1 by Edward A. Lee and Christopher Hylands (see: http://ptolemy.eecs.berkeley.edu/java/ptplot), but until now this has not been utilized.
The programming of the graphics is best explained by means of an example. Let's assume that we want to add a third plotting screen to the existing program. This is what has to be done:

1) In the class "PlotChoicePanel" the line

   ```
   plotCheckBox  = new Checkbox [3];
   ```

   has to be changed into

   ```
   plotCheckBox  = new Checkbox [4];
   ```

   and the additional checkbox has to be initialized and added to the panel according to the way being used for the first three checkboxes. This way we are setting the stage to obtain control over the new plot screen we are about to generate.

2) In the class "PlotChoicePanel" the method handleEvent() must be enhanced by

   ```
   else if(e.target == plotCheckBox[3])
   {
    if (plotCheckBox[3].getState() == true)//new plot screen
    {
      deScreen.newPlotScreen2();
      deScreen.plot_screen2_exists = true;
    }
    else // disable plot
    {
   ```

```
    deScreen.destroyPlotScreen2();
    deScreen.plot_screen2_exists = false;
  }

}
```

so that a click on the new checkbox has the proper effect. As we can see from the code lines above we have introduced some variables and methods in class “DEScreen” that have not existed before. These are further explained in the next step:

3) The class “DEScreen” must be furnished with the following variables and methods:
a) the object
`public PlotScreen   plotScreen2;`
must be introduced. This is the new plot screen which shall appear when the pertinent checkbox is clicked.

b) the variable
`public boolean plot_screen2_exists = false;`
must be introduced. It remembers whether the screen object exists or not. This variable helps to generate and destroy the plot screen at any time during the optimization.

c) The methods newPlotScreen2() and destroyPlotScreen2() must be added analogously to the already existing methods of the same kind. This means among other things that the instantiation of the new plot screen has to take care of the plot screen number “2” via
`...new PlotScreen(this,2);`

d) `repaint()` must be augmented with an if-clause which takes care of `plotScreen2`.

4) In class “PlotScreen” we define a new plot graph according to
`public PlotGraph2   plotGraph2;`
and do the appropriate changes in the constructor of “PlotScreen”, i.e. give the new plot screen a title and instantiate it when the graph_type variable is set to 2. The method refreshImage() has also to be updated accordingly.

5) Now we finally write a new class PlotGraph2.java similiar to the plot graphs that have been written before. While all the previous steps dealt primarily with the GUI control this class goes to the heart of the matter: the plot routine itself. Let’s assume that we don’t want to change the basic way of plotting which is having a coordinate system with certain minimum and maximum abscissa and ordinate values, and plotting some graph by connecting samples of the graph via straight lines.

In analogy to the already existing PlotGraph classes we define min_x, min_y, max_x, max_y in the method initParameters(). We can also change the tics by adapting x_tics and y_tics if we want to.

The program defines a static part of the graphics which doesn’t change (as e.g. the coordinate system) and a dynamic “offscreenGraphics”. The static part is handled in the method preparePlot() which again is called in initGraphics(). The method initGraphics() is called in init() which again is called in the paint() and refreshImage() method, and there the chain of calls end. Except for preparePlot() all these methods can remain untouched. Currently preparePlot contains the method call plotAxes() which is a method that has to be adapted to the problem at hand.

The dynamic and most important part of the plotting is handled in the method plot() which is called in refreshImage(). Method plot() makes use of the method drawLine() which connects two points with a single solid hairline. In order to be able to work in the relative coordinate system defined by min_x, min_y, max_x, max_y, the methods absX() and absY() are provided. The latter transform the relative coordinate values into absolute ones.

2. References

[Cha97] Chan, P. and Lee, R., The Java Class Libraries - An Annotated Reference, Addison-Wesley, 1997.

[Cod97] Coad, P. and Mayfield, M., Java Design - Building Better Apps & Applets, Yourdon Press, 1997.

[Dav96] Davis, S.R., Learn Java Now, Microsoft Press, 1996.

[Gam95] Gamma, E. et alii, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[Lem96] Lemay, L., Perkins, C.L. and Morrison, M., Teach Yourself Java in 21 Days, Sams.net Publishing, 1996.

[SP96_1] Storn, R. and Price, K., Minimizing the real functions of the ICEC'96 contest by Differential Evolution, IEEE Conference on Evolutionary Computation, Nagoya, 1996, pp. 842 - 844.

[Sto96_1] Storn, R., On the Usage of Differential Evolution for Function Optimization, NAFIPS 1996, Berkeley, pp. 519 - 523.

[SP96_2] Price, K. and Storn, R., Differential Evolution: Numerical Optimization Made Easy, Dr. Dobb's Journal, April 97, pp. 18 - 24.

[SP96_3] Storn, R. and Price, K., Differential Evolution - a Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimization, Kluwer Academic Publishers, 1997, Vol. 11, pp. 341 - 359.