MooN

A Framework for Metaheuristical Optimization

Developers Manual

by
PG 431*

*At the University of Dortmund every Computer Science student has to attend a year long project, the "Projektgruppe". The project in which MooN was developed was project 431, thus "PG 431". The members of the project are (in alphabetical order):

Selcuk Balci, Sören Blom, Daniel Blum, Vedran Divkovic, Dirk Hoppe, Djamila Lindemann, Ulf Schneider, Bianca Selzam, Thomas Tometzki, Marko Tosic, Igor Vatolkin and Stefan Walter

http://ls11-www.cs.uni-dortmund.de

Contents

1 Introduction

1.1 About MooN

The field of (meta)heuristical optimization is very dynamic. Although a wide variety of optimization algorithms already exist, many new ideas appear regularly and are transformed into applicable algorithms. Important questions in this area concern the performance on special test problems and the configuration of heuristic relevant parameters to improve performance.

MooN is an application whose aim is to simplify working in this area. Because it has an extendable structure, heuristics or test problems can easily be integrated as plug-ins. Therefore the effort and time to implement new (meta)heuristics is minimized. Once plug-ins are installed, they can be combined in arbitrary ways so that research and testing is easily possible.

The MooN application is shipped under the GNU General Public License (see appendix B). This allows anyone to extend the source code to his or her own needs and share the results with anyone interested, as long as the change remains GPL licensed. This way there is no licensing barrier when extending MooN.

1.2 About this Manual

This manual is written for anyone who is interested in the internal structures of MooN. There should be primarily two groups of people that have such an interest: MooN plug-in developers and MooN application developers.

The first group are people who want to extend MooN via its flexible plug-in approach, e.g. implementing a new heuristic or problem. Thereby improving the range of situations MooN can be used as a tool for in (meta)heuristical optimization and research. The second group is changing parts of the internal structure of MooN with the intent to improve the overall performance, the user interface, logging or other general aspects of the application.

The content of this manual is divided into three sections. The first section provides general information about the application and the manual, the second section discusses details of the applications implementation and the design ideas behind it and section 3 is a guide to write plug-ins for MooN.

Plug-in developers can safely skip section 2 since the knowledge of the internals is not necessary to write plug-ins for MooN. It is assumed that the reader is familiar with handling MooN as described in the user manual which is also part of the documentation package of the MooN distribution.

1.3 The General Structure of MooN

MooN is separated into three different layers (fig. 1): *core*, *user interface* and *plug-ins*. The core itself is separated in several modules, which will be described

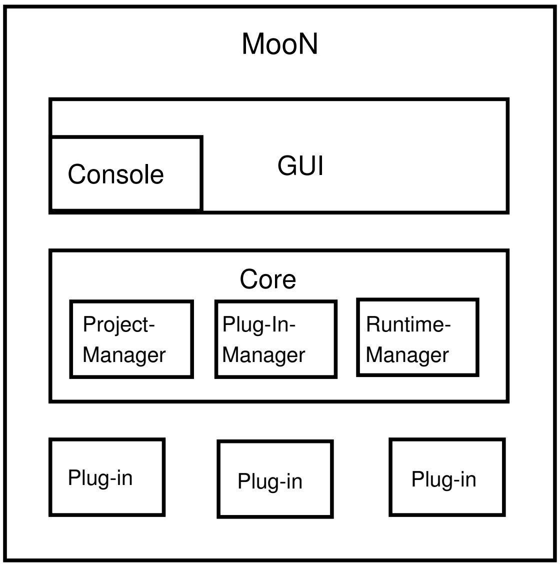

Figure 1: Layered structure of MooN

in section 2. Plug-ins can be of three different types: heuristics, problems and exit conditions.

With the separation into three layers we achieve a separation of the different concerns:

- The user interfaces take care of the presentation of the functionality that the core provides. Since it is totally separated, many different types of user interfaces can be implemented independently from each other.
- The core has the main functionality to manage the installed plug-ins, to connect and configure them in user projects and to execute them.
- The plug-ins encapsulate specific domains (optimization heuristics or application problems). Due to the API that the core provides, implementation is straightforward which allow to have a tool which is both comfortable and flexible at the same time.

2 The Application

2.1 Plug-In Manager

As described in section 1.3, MooN is separated into three parts: user interfaces, core and plug-ins. The plug-in manager is the part of the core which administers the plug-ins; that is, installs, removes and loads them at the program start. In the following section we describe in detail how the plug-in manager deals with those tasks. For a description on how to write plug-ins see section 3.1.

The plug-in manager is mainly constituted of two classes, which reside in the package `moon.core.pluginManagement`:

- `Plugin`: The class that represents an installed and loaded plug-in in MooN.
- `PluginManager`: The class providing all relevant methods to administer plug-ins. It also has a 'registry' of available (i.e. loaded) plug-ins.

2.1.1 Installation of Plug-ins

In order to use a new plug-in in MooN, it needs to be installed. Since a plug-in is merely a `.jar`-file, "installing" basically means unpacking the archive into a special plug-in directory[1]. The `PluginManager` offers the method:

```
PluginManager.installPlugin(String pathToJar).
```

Every plug-in is installed into its own directory which is named after the `.jar`-file it was installed from. This folder name is used as the plug-in name in the application once it has been loaded. For example, a plug-in that has been installed from a file `fooBar.jar` would be installed in a folder called `fooBar`, which would also be its name in the MooN later on. The only requirements for a plug-in in terms of installation are that it ends with `.jar` and that it is a valid Jar archive. Right after successful installation of a plug-in the plug-in manager would attempt to load it (see below), so it is immediately available for use in the running application.

2.1.2 Loading Plug-Ins

In order to give the user the flexibility to choose which plug-ins to use, MooN needs the ability to integrate classes into the program at runtime. Using Java, it is easy to implement this functionality using "ClassLoader" from the Java Library. Since the plug-in approach should give the developer the freedom to implement a plug-in with arbitrarily many classes, MooN needs to be able to identify the "main class" of a plug-in, where "main class" means that it implements one of the plug-in interface classes in `moon.interfaces`. This class needs to be specified in the file

[1]The default is to install plug-ins below the `plugins` folder in the MooN installation folder.

`plugin.properties` which is a required part of every plug-in (also section 3.2). The loading is done by the method

`PluginManager.loadPlugin(File pluginDirectory, String pluginName),`

where `pluginDirectory` points to the directory in which MooN holds all installed plug-ins in, and `pluginName` is the name (therefore the folder name) of the plug-in which to load. The method searches all class files in the plug-in folder, which, by convention, are all files ending with `.class`, and checks if the ”main class” specified in the `plugin.properties` points to a class that was found before.

If the class is found all the classes need to be made known to the Java Virtual Machine, so they can be instantiated later on by the application. This can conveniently be done by using a *class loader*.

We chose to use `java.net.URLClassLoader`. All class files that had been found earlier are loaded that way. The final step is to create a plug-in object and to register it with the plug-in managers registry.

2.1.3 Removal of Plug-Ins

If a plug-in is not needed anymore the user has the possibility to remove it. The corresponding method is:

`PluginManager.removePlugin(String pluginName).`

If a plug-in exists under that name, it will be deregistered from the plug-in manager's registry and the folder will be removed as well. As stated before, the plug-in name and folder name are the same.

2.1.4 Using Plug-ins in the Application

To make plug-ins easy to use for the user and the developers of other MooN components at the same time, both `PluginManager` and `Plugin` offer many methods to get information about plug-ins and parts of plug-ins in a straightforward manner.

The plug-in registry, for instance, offers methods to retrieve a list of all loaded plug-ins, and also to retrieve all plug-ins of a certain type (which are Heuristic, Problem and ExitCondition), while certainly a single plug-in can be retrieved by its name.

The `Plugin` class has methods to retrieve the “main” class and informations which are useful for other components such as the Graphical User Interface, the project manager or the runtime.

2.2 Project Manager

The main functionality of MooN is the optimization on a heuristic-problem combination, called a *single run*. Since any serious research will include a large number of these single runs, MooN allows the user to maintain “projects” called *complete runs*. Complete runs are XML files which group arbitrary numbers of single runs.

The structure of these files (see section 2.2.1) is validated against an *XML Schema Description* (XSD), which constitutes a grammar for valid complete runs. With this schema it is easy to verify that the files have all of the necessary information to describe a project in MooN.

To load (section 2.2.2) and save (section 2.2.3) complete runs, the class `ProjectManager` in the package `moon.core.projectManagement` is used.

2.2.1 The Structure of “complete run” XML Files

In this section we describe the minimal set of elements necessary to create a valid complete run document. A complete and valid example of a complete run XML file can be found in appendix A.1. The authorative resource for details is the `moon.xsd` XSD document itself. It can be found in the source distribution of MooN in the package `moon.projectmanagement` and at the end of this document (appendix A.2).

Each complete run XML file may only contain one `<completeRun>` element. Since the complete run merely groups single runs it can have (arbitrarily) many `<singleRun>` child elements. A complete run may contain a `<description>` tag with information about the run. To this point, the (incomplete) document would look like this:

```
<completeRun>
    <description>
        Some description...
    </description>
    <singleRun>
        ...
    </singleRun>
        ...
    <singleRun>
        ...
    </singleRun>
</completeRun>
```

A `<singleRun>` element has at least four children. A `<description>` tag may be provided, similar to the one under `<completeRun>`:

```
<singleRun>
    <description>
        Some description...
    </description>
    <heuristic>
        ...
    </heuristic>
    <problem>
        ...
    </problem>
        ...
    </problem>
    <exitCondition>
        ...
    </exitCondition>
    <outputHandler>
    </outputHandler>
</singleRun>
```

These elements correspond to the different classes a single run consists of. The first three (heuristic, problem and exit condition) are structured the same way. In the XSD they inherit it from a common description. As they constitute the core of the plug-in approach, they hold a reference to the plug-in main class (see section 3.2) they represent and a list of parameters for this class. In the XML file the `<plugin>` and `<class>` elements are mandatory, whereas the `<parameters>` elements with its `<parameter>` tags are optional depending on whether or not the plug-in has parameters.

In this example, a imaginary heuristic is used to show the different elements. The XML tags are the same as in ”problem” or ”exit condition”.

```
<heuristic>
    <plugin>MyPlugin</plugin>
    <class>com.foobar.MyPlugin.MyMainClass</class>
    <parameters>
        <parameter key="prop1" value="10"/>
        <parameter key="prop2" value="low"/>
              ...
    </parameters>
</heuristic>
```

The `<outputHandler>` configures the logging component of MooN. Logging can happen for different “categories” (depending on what the heuristic in the corresponding single run offers) and for each category a file name can be specified. Finally, a category does not have to write its output for every iteration, so the frequency is given by the logging interval.

```
<outputHandler>
 <category>
     <filename>cat1.log</filename>
     <name>cat1</name>
     <interval>1</interval>
 </category>
     ...
 <category>
    ....
 </category>
</outputHandler>
```

2.2.2 Loading Complete Runs

Loading a complete run needs to cope with two things:

1. Parsing (reading) the XML file.
2. Creating the respective objects from the information found in the XML file.

Those two tasks are done in an interleaving manner in the class `XMLReader`. Since we have the XSD and use a validating parser (Xerces) it can be assumed that only valid documents will be considered. Using JDOM it is easy to access the DOM tree and to extract the necessary informations.

The loading procedure is the main point of interaction between the project manager and the plugin manager. The `XMLReader` identifies the plug-ins specified in the complete run and queries the `PluginManager` over them. In case a plug-in cannot be found (i.e. because it is not installed) an `UnableToLoadXMLException` will be thrown and the loading will be aborted. The same is true if any information about an existing plug-in is incorrect, i.e. wrong parameters have been used.

If the loading succeeded a `CompleteRun` object will be passed to the runtime component (see section 2.3) for further processing. This object represents the XML file in so far that it contains a number of `SingleRun` objects that contain the completely configured plug-in classes.

2.2.3 Storing Complete Runs

Storing XML files is a functionality that is primarily needed by the GUI. In the GUI the user directly manipulates the complete and single run objects described in section 2.2.2. It is the therefore necessary to be able to preserve their state into a complete run XML file.

In analogy to the `XMLReader` from above, the `XMLWriter` does the job of writing complete run files. It collects the information from the objects and inserts them into the XML file at the correct position.

2.3 Runtime

Now that the plug-ins (section 2.1) and the creation of projects in form of complete runs (section 2.2) have been described it is time to introduce the component which ties it all together, the *runtime*. For execution of the complete run, the runtime accesses the `CompleteRun` object created by the project manager.

The runtime component's classes are `RuntimeManager` and `RuntimeExecuter`. The first is a wrapper class accessed by the GUI and the command line tool. It loads a complete run from the XML file by involving the project manager. The second class does the actual job of preparing the individual single runs, executing them and cleaning up afterwards. Due to our abstraction every plug-in contained in a single run goes through those three stages (see also section 2.4).

2.3.1 Initializing the Run

To avoid confusion it might be helpful to underline that the initialization of a run, as described in this section, is not the same as initializing the plug-ins as mentioned in section 2.2. At the time the runtime manager enters the stage, the technicalities have been dealt with, classes have been loaded and the plug-ins are configured in so far as they have had their parameters set.

Initializing a single run means to call the `Heuristic.initialize()` method. In this method the plug-ins prepare everything they need to be ready to run immediately. To see why this method is necessary even next to the class' constructor, let's consider the following example: [2]

> A problem plug-in needs to be written to include some third-party software which implements the objective function of the problem. It is written in C and can only be accessed over the network so the MooN plug-in functions as a wrapper. Due to restrictions of the other software only one connection can be established at a time and it times-out after a certain while.

In this situation the use of the initialization method becomes obvious: It would not be possible to establish the connection in the constructor. With more than one single run including this problem plug-in that would be created by the project manager the constraint of "only one connection at a time" would be violated. In addition establishing a connection during object creation would probably result in a timeout since there is no telling when and if the single run will be executed in the near future.

After the init method has been called the runtime sets the heuristic's problem and its output handler. This is, again, not the loading and initialization of the corresponding plug-ins, but their close-to-execution-time setup.

[2]The situation in this example appeared during the project. The resulting plug-in couldn't become part of MooN due to license issues.

2.3.2 Executing the Run

From the runtime's perspective the execution of a single run comes down to trigger the heuristic specified in the run, since the interaction between heuristic and problem is done by the heuristic which takes care of having search points evaluated by the problem. The `RuntimeExecutor` calls the `Heuristic.nextGeneration()` method to accomplish that.

2.3.3 Cleaning Up Afterwards

After all single runs have been executed it is important to finish up in a sensible manner. As seen above it might be necessary to close network connections, flush and close (file) output streams or do other work. The plug-ins should include any such functionality into the `Heuristic.cleanUp()` method. The runtime manager calls this method for the heuristic, the problem and the exit condition.

Additionally it informs the output handler that the logging is finished (via `OutputHandler.close()`) so that all output streams that the output handler was writing into can be flushed and closed.

2.4 The Framework API (Interfaces)

2.4.1 The Overall Package Structure

The presentation in this manual does not cover the complete MooN application programming interface (API). A complete description of all classes and methods can be found in the `doc/javadoc` folder of any MooN installation. Instead we will show the overall structure of the packages in MooN and focus on the organization of the `interface` package especially.

The package structure of MooN can be seen in figure 2. Four main packages which encapsulate the different aspects of the application:

1. `core`: It includes the main components of the application (property, runtime and plugin manager).
2. `interface`: Defines the layer of abstraction between the core and the plug-ins.
3. `gui`: Has the code for the graphical user interface.
4. `plugin`: Contains plug-ins that were implemented by PG 431 during the development of MooN.

2.4.2 The `interface` Package

Due to our approach the plug-in plays the central role in the organization of code. Therefore the interface `Plugin` is a central part of our API. It contains

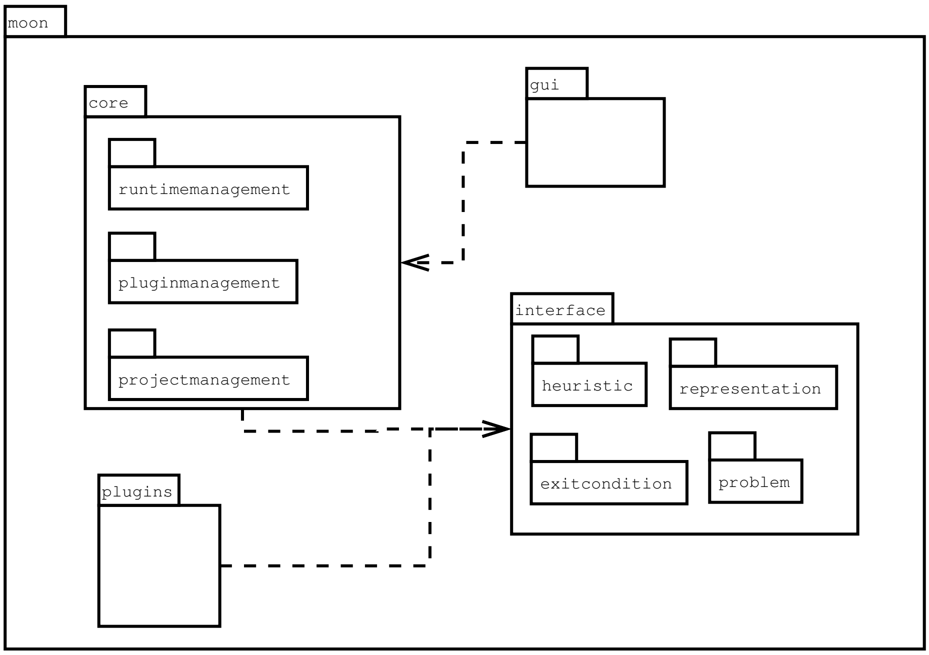

Figure 2: Package structure of MooN

important methods for plug-in management (section 2.1) and represents the plug-ins in MooN. Since plug-ins can be very different, there exist specializations as shown in figure 3 which follow the three types of plug-ins described throughout this manual.

The methods provided by the interface are important for communicating configuration details between MooN and the plug-in (`getParams()`, `setParams()`, etc) [3] as done by the project manager or runtime related items (`initialize()`, `cleanUp()`.

It is imaginable that there will be other plug-in types in the future, e.g. for visualization. The guideline for introducing plug-ins from our point of view was to keep domain specific code out of the core. Something like problem specific visualization would therefore be seen as a non-core feature.

2.4.3 . . . and Its Subpackages

The `interface` package has five subpackages. The packages `heuristic`, `problem` and `exitCondition` were introduced as a consequence of having these three specializations of the `Plugin` interfaces. We will look at those three first, describing the main interfaces. Figure 4 shows the methods of the aforementioned interfaces. Although they are depicted next to each other, they reside in three individual subpackages.

[3] for a thorough description see section 3.1

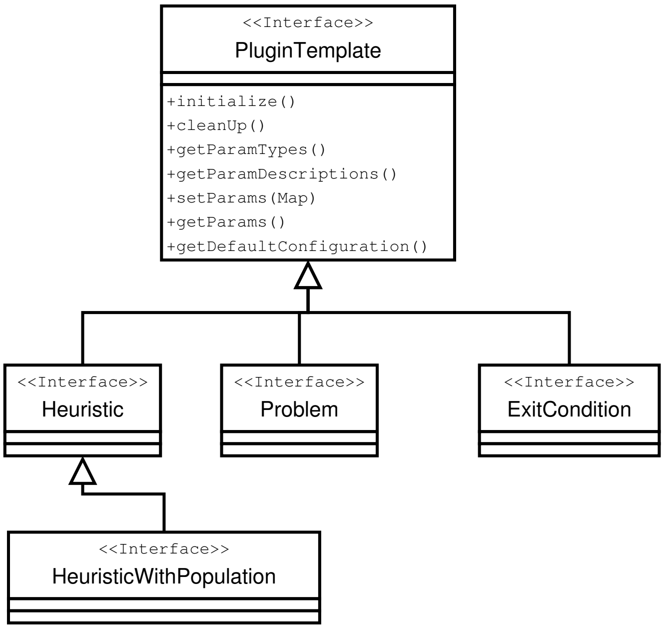

Figure 3: Specialization Hierarchy of interface `PluginTemplate`.

`moon.interfaces.problem` `Problem`'s only method is `getFitness(Solution)` which evaluates a given `Solution`. It can be seen that the problem has no knowledge of the heuristic or exit condition, since it only communicates via solutions that are generated by a heuristic.

`moon.interfaces.exitCondition` The `ExitCondition` interface also has a single method only, `mustTerminate(Heuristic)`, which examines whether or not a heuristic needs to terminate. It was necessary to find a rather general interface (a complete heuristic, not only a population) since MooN wants to cover a wide range of possible heuristics and several of them are not population based.

`moon.interfaces.heuristic` `Heuristic` in contrast has a number of methods which can be grouped by function

- Runtime related:
  - `nextGeneration()`
  - `getGenerations()`
- Visualization related:
  - `getGlobalBestIndividual()`
  - `getCurrentBestIndividual()`

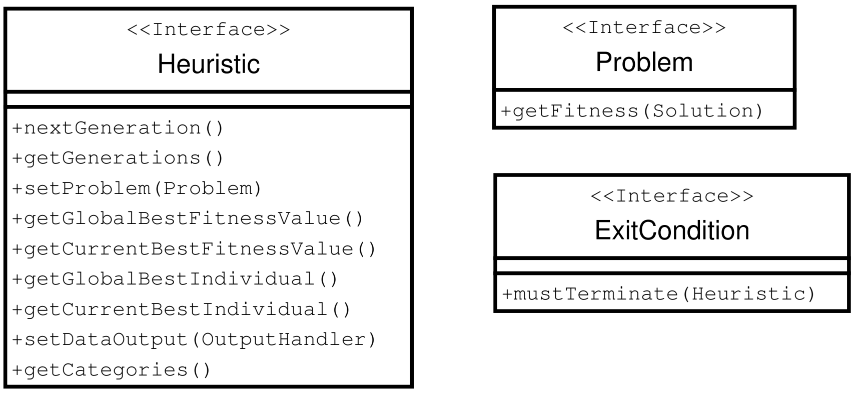

Figure 4: `Plugin`-derived subinterfaces with their methods.

  - `getGlobalBestFitness()`
  - `getCurrentBestFitness()`

- Output related:
  - `setDataOutputHandler()`
  - `getCategories()`

The most interesting method of this list is `Heuristic.nextGeneration()`, discussed in detail in section 2.3. Next to `Heuristic` this package contains other interfaces and classes worth mentioning: `HeuristicWithPopulation` models the typical population based heuristic together with `Population` and `Individual`.

`moon.interfaces.representation` One aspect in terms of (meta)heurstic design has not been covered by the interfaces so far: Solutions and fitness values. To increase interoperability and code reusage, we introduced general interfaces `Solution` and `FitnessValue` and a variety of specializations. `Solution` for instance has five implementing classes. Since the whole interaction between plug-ins is based on these representations, every implementation of solutions or fitness values must implement those interfaces.

`moon.interfaces.exception` Compared to the other four, `exception` is just a container package for all exceptions that are related with the implementation of plug-ins. There exist exceptions for all types of plug-ins (`ExitConditonException`, `ProblemException`, `HeuristicException` all inheriting from `PluginException`), an Exception that can occur during Logging (`LoggingException`) and others.

3 Extending MooN

3.1 Writing Plug-Ins

3.1.1 General Plug-Ins

All plug-in types, no matter if heuristic, problem, or exit condition, have in common that their parameters can differ in name, number, and types. But for all plug-ins some basic operations concerning their parameters are needed. For example the transfer of the parameter configuration into an XML file and back into the application requires a flexible interface, which can deal with a variable number of parameters, which themselves can be of different types. These requirements are served by the data structure `java.lang.Map`, which is used as the transporter throughout the interface. The `Map` is variable in its size and allows key value pairs of objects. As keys `java.lang.String` objects of the respective parameter names are used. Parameter names have to be unique within a plug-in.

It is a good idea to introduce constants for the parameter keys, as each are used several times. Suppose one parameter is used in the plug-in:

```
private int alpha;
```

Then we define also:

```
private static final String PARAM_KEY_ALPHA = "alpha";
```

Now the parameter related methods in detail:

The method `setParams(Map)` is invoked when someone has created a single run with this plug-in via GUI and then accepts the parameters from the default or his own configuration. When the project management loads an XML file and creates the related complete run object, the parameters stored in the XML structure are placed in the associated plug-in objects via this method as well.

```
public void setParams(Map params) throws PluginException {
    try {
        alpha = ((Integer)params.get(PARAM_KEY_ALPHA)).intValue();
        beta  = ((Float)params.get(PARAM_KEY_BETA)).floatValue();
        ...
    } catch (ClassCastException e) {
        throw new PluginException(e);
    } catch (NullPointerException e) {
        throw new PluginException(e);
    }
    // recommended, for easier implementation of getParams()
    paramsMap = params;
}
```

The method Map getParams() is used when the project manager writes a complete run object into an XML file to obtain the parameters set in this plug-in. The method is also used by the GUI to fill the entry fields for parameters with the already set parameters.

```
public Map getParams() {
   // requires storage of the Map in setParams(Map), recommended
   return paramsMap;
}
```

The method getDefaultconfiguration() is used by the GUI to fill the parameter entry fields with sensible values when new plug-ins are chosen. The implementation is straightforward: create a new HashMap and place a pair for every parameter as follows in it. The key is as usual the constant describing the parameter and the value is an object with the default value.

```
public Map getDefaultConfiguration() {
   HashMap defaultMap = new HashMap();
   defaultMap.put(PARAM_KEY_ALPHA, new Integer("123"));
   defaultMap.put(PARAM_KEY_BETA, ...);
   ...
   return defaultMap;
}
```

The method Map getParamTypes() is used by the GUI and the XMLReader, when parameter values given as Strings (in XML file or GUI entry field) are put in the Map as objects from the parameter's type. It returns each parameter's type as a Class object. With the information from this method it is possible to instantiate objects of the right type, using their constructor taking a String as the argument.

```
public Map getParamTypes() {
   HashMap types = new HashMap();
   types.put(PARAM_KEY_ALPHA, Integer.class);
   types.put(PARAM_KEY_BETA, ...);
   ...
   return types;
}
```

The method Map getParamDescriptions() is used in the GUI to create a tool tip when the user places the mouse pointer over a parameter's entry field. This way more information about the parameter is provided. Keys are the constants and the values are Strings with the description.

```
public Map getParamDescriptions() {
   HashMap descriptions = new HashMap();
   descriptions.put(PARAM_KEY_ALPHA, "Parameter alpha ...");
   descriptions.put(PARAM_KEY_BETA, ...);
   ...
   return descriptions;
}
```

These were the parameter related methods in the `PluginTemplate` interface. There are two more important methods specified regarding the life cycle: A plug-in is *created* and its parameters are set via `setParams(Map)`. This can happen during run creation or editing. When runs are executed, the plug-in is initialized before it executes its particular work. For this, there are more specific interfaces responsible, as described below. At the end it is possible to finish the object's life cycle in a proper way.

The method `initialize()` is used to prepare the plug-in for its application. For example, this could entail establishing connections to external servers if needed, opening files with further data, as specified in some parameters or initializing populations needed in heuristics, etc. The method is invoked by the runtime just before run execution.

```
public initialize() {
   ...
}
```

The method `cleanUp()` is invoked by the runtime after the run execution is finished. It is a good point to close files or connections, or do whatever is appropriate to ”clean up”.

```
public cleanUp() {
   ...
}
```

3.1.2 Heuristic Plug-Ins

”Heuristic” is the most important plug-in type in the MooN architecture. There are three groups of methods: the optimization related, the information and the logging methods.

The methods for the working process of a heuristic are `setProblem(Problem)` and `nextGeneration()`. The first is called before `initialize()` and has the already initialized problem plug-in as parameter. That way the heuristic always has a valid reference to the problem that it will try to optimize later.

```
public void setProblem(Problem problem) {
   this.problem = problem;
}
```

The latter method, `nextGeneration()`, is the main part of every heuristic. It is invoked repeatedly after `initialize()` by the runtime until the exit condition is reached. In this method the heuristic specific computation of the optimization process is implemented.

```
public void nextGeneration() throws HeuristicException {
    ...
}
```

As the state of the heuristic is of interest during execution, several methods exist to obtain relevant parts of the state. *best fitness values*, *best individuals*, as well as the *number of generations* performed are more common examples. These methods are used by exit conditions and for the runtime visualization:

```
public int getGenerations();
public FitnessValue getGlobalBestFitnessValue();
public FitnessValue getCurrentBestFitnessValue();
public Individual getGlobalBestIndividual();
public Individual getCurrentBestIndividual();
```

The logging of runtime data beyond the scope of the above methods is performed by the two methods from the `Heuristic` interface.

The method `getCategories()` is used by the GUI to retrieve the categories for configuration to the user. The retrieved `Map` object contains entries with categories as keys and descriptions as their values.

```
public Map getCategories() {
    HashMap categories = new HashMap();
    categories.put(SPECIAL_CATEGORY,
            "Special information from this heuristic.");
    categories.put(OTHER_CATEGORY, "...");
    ...

    return categories;
}
```

The output handler used by the heuristic to do the logging is set by the method `setDataOutput(OutputHandler)`. This happens before `initialize()` is called.

```
public void setDataOutput(OutputHandler handler) {
   myOutputHandler = handler;
}
```

The logging is usually done after each generation was computed. It might be convenient to delegate this task to a separate method. The output handler stores the logging intervals. The heuristic calls `isLogIteration(int,String)` to determine if the particular category is to log in the current generation. If this is the case, the output handler receives the category name and related log information.

```
private void doTheLogging() throws LoggingException {

    if (myOutputHandler.isLogIteration(currentGeneration,
                                       SPECIAL_CATEGORY)){
            myOutputHandler.write(SPECIAL_CATEGORY,
                          "<The special informations>");
    }
    ...
}
```

3.1.3 Problem Plug-Ins

The problem plug-in has only one special method. It is called by heuristics, which receive fitness values for their produced solutions:

```
public FitnessValue getFitness(Solution aSolution)
                                   throws ProblemException;
```

3.1.4 Exit Condition Plug-Ins

The exit condition plug-in is used by the runtime to determine when a single run needs to terminate its execution. Via the method `mustTerminate()` the exit condition receives a heuristic object which is examined to decide if the exit condition is met.

```
boolean mustTerminate(Heuristic heuristic)
                                throws ExitConditionException;
```

3.2 The `plugin.properties` File

In order to use a plug-in one has to provide additional information next to the plug-in's code. The file where this information is stored is called `plugin.properties`. It has to reside in the main-folder of the plug-in and provide a required set of plug-in details.

3.2.1 plugin.properties-Template

First we will supply a template for the plugin.properties-file and describe the meaning of the entries below.

```
# Property File for <name of plug-in>
#
# <optional short description of Plugin>
#

# PLUGIN_TYPE can be either "heuristic" or "problem"
# or "exitCondition"
PLUGIN_TYPE=<heuristic/problem/exitCondition>

# PLUGIN_DESCRIPTION gives a short overview of the plugin
PLUGIN_DESCRIPTION=<short description>

# PLUGIN_MAIN_CLASS points to the class that implements an
# extension of PluginTemplate
PLUGIN_MAIN_CLASS=<mainclass>
```

3.2.2 Options

Of the three properties shown above PLUGIN_TYPE and PLUGIN_MAIN_CLASS have to be set, PLUGIN_DESCRIPTION can be left empty. Comments have to be preceded by a # as first character of each line.

```
PLUGIN_TYPE=<heuristic/problem/exitCondition>
```

Here the type of the plug-in is given by specifying one of the three types heuristic, problem or exitCondition.

```
PLUGIN_DESCRIPTION=<short description>
```

A short description of the plug-in capabilities has to be added here. The text given here will only be used in the GUI to explain users of the plug-in the general behavior of the plug-in.

```
PLUGIN_MAIN_CLASS=<mainclass>
```

The main class is the class that extends moon.interfaces.PluginTemplate. The fully qualified name of the class including the package name needs to be given (e.g. moon.plugin.heuristic.pso.StandardPSO).

3.3 Packing Plug-Ins

When a plug-in is implemented, the last step is to create a `jar`-file for deployment. The file consists of the classes used by the plug-in, the `plugin.properties` file as described in the previous chapter and optional other files used by the plug-in. The `plugin.properties` file has to be placed in the root path of the `jar`-file. The classes have to be in directories according to the fully qualified class name (e.g. class `moon.plugin.MyPlugin` in `moon/plugin/`). This `jar`-file now contains everything needed, so that it can be distributed and used by any MooN user for installation of the plug-in.

3.4 Installation

The plug-in's installation procedure was designed to be easy to apply for the user. Plug-ins can be installed in two different ways: Using the GUI or using the MooN command line tool.

3.4.1 Installation Using the GUI

After starting MooN File/Install plug-in has to be selected in the menu bar. A file dialog will be shown and the new plug-in has to be selected (e.g. `myPlugIn.jar`). Afterwards the plug-in will be installed. If this was successful the plug-in is ready to use.

3.4.2 Installation Using the Console

From the command line `MooN` has to be started with parameter `-i` followed by the name of the plug-in (e.g. `moon -i myJars/myPlugIn.jar`) to achieve the same as above.

We hope this manual will be found useful in extending and enhancing MooN.

PG 431

A Complete Run Details

A.1 Complete Run Example

The following example shows a valid complete run. It is executable provided the used plug-ins are installed. The only manipulation which might be problematic for the parser is the `class` tag of plugin `FitnessThreshold` since the long class name had to be cut into two lines for layout's sake.

```
<?xml version="1.0" encoding="UTF-8"?>
<completeRun xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="../projectManagement/moon.xsd">
  <description>Quite a fancy complete run description</description>
  <singleRun>
    <description>First single run is doing this and that</description>
    <problem>
      <plugin>Rastrigin</plugin>
      <class>moon.plugin.problem.rastrigin.Rastrigin</class>
      <parameters />
    </problem>
    <heuristic>
      <plugin>SA</plugin>
      <class>moon.plugin.heuristic.sa.SimulatedAnnealing</class>
      <parameters>
        <parameter name="Solution minimum" value="-10.0"/>
        <parameter name="Start temperature" value="10.0"/>
        <parameter name="Reduction factor" value="0.9"/>
        <parameter name="Dimension" value="6"/>
        <parameter name="Maximum try" value="100"/>
        <parameter name="Solution maximum" value="10.0"/>
      </parameters>
    </heuristic>
    <exitCondition>
      <plugin>Fitness_Threshold</plugin>
      <class>moon.plugin.exitCondition.exitConditionFitnessThreshold.
            ExitConditionFitnessThreshold</class>
      <parameters>
        <parameter name="Lower bound" value="true"/>
        <parameter name="Threshold" value="0.0"/>
      </parameters>
    </exitCondition>
    <outputHandler>
      <options multipleLogging="false" repetitions="1"/>
      <category name="global_best_individual" intervals="1"
                filename="Default.log"/>
      <category name="current_best_individual" intervals="1"
                filename="Default.log"/>
    </outputHandler>
```

```
  </singleRun>
  <singleRun>
    <description>
         The second one tries the same heuristic on a different problem
    </description>
    <problem>
      <plugin>tspProblem</plugin>
      <class>moon.plugin.problem.tspProblem.TspProblem</class>
      <parameters>
        <parameter name="Nodes informations file" value="berlin52.para"/>
        <parameter name="Nodes number" value="52"/>
      </parameters>
    </problem>
    <heuristic>
      <plugin>SA</plugin>
      <class>moon.plugin.heuristic.sa.SimulatedAnnealing</class>
      <parameters>
        <parameter name="Solution minimum" value="-10.0"/>
        <parameter name="Start temperature" value="10.0"/>
        <parameter name="Reduction factor" value="0.9"/>
        <parameter name="Dimension" value="6"/>
        <parameter name="Maximum try" value="100"/>
        <parameter name="Solution maximum" value="10.0"/>
      </parameters>
    </heuristic>
    <exitCondition>
      <plugin>Fitness_Threshold</plugin>
      <class>moon.plugin.exitCondition.exitConditionFitnessThreshold.
             ExitConditionFitnessThreshold</class>
      <parameters>
        <parameter name="Lower bound" value="true"/>
        <parameter name="Threshold" value="0.0"/>
      </parameters>
    </exitCondition>
    <outputHandler>
      <options multipleLogging="false" repetitions="1"/>
      <category name="global_best_individual" intervals="1"
                filename="Default.log"/>
      <category name="current_best_individual" intervals="1"
                filename="Default.log"/>
    </outputHandler>
  </singleRun>
</completeRun>
```

A.2 The moon.xsd File

This is the XSD file for the complete runs. It is part of the (source) distribution of MooN and can be found under `src/moon/core/projectmanager`. For a detailed description see 2.2).

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    moon.xsd

    This is the schema for the project description which defines
    CompleteRuns for MooN.
    Basically a CompleteRun consists of a number of SingleRuns
    which have at least one heuristic, one problem and one exit
    condition.
    Additionally the OutputHandler can be configured for each
    SingleRun.

    Created on 19.11.2003

    PG431 - MooN, a framework for metaheuristical optimization
    Copyright (c) 2003, 2004 Projektgruppe 431:
    Selcuk Balci, Sören Blom, Daniel Blum, Vedran Divkovic, Dirk
    Hoppe, Djamila Lindemann, Ulf Schneider, Bianca Selzam, Thomas
    Tometzki, Marko Tosic, Igor Vatolkin, Stefan Walter

    University of Dortmund
    Department of Computer Science
    Chair of Systems Analysis (LS 11)
    44221 Dortmund
    Germany
    http://ls11-www.cs.uni-dortmund.de

    This program is free software; you can redistribute it and/or
    modify it under the terms of the GNU General Public License
    as published by the Free Software Foundation; either version 2
    of the License, or (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public
    License along with this program; if not, write to the Free
    Software Foundation, Inc., 59 Temple Place - Suite 330, Boston,
    MA  02111-1307, USA.
-->

<xs:schema xsi:schemaLocation="http://www.w3.org/2001/XMLSchema
```

```
        http://www.w3.org/2001/XMLSchema.xsd"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="completeRun">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="description" type="xs:string" minOccurs="0"/>
         <xs:element name="singleRun" maxOccurs="unbounded">
           <xs:complexType>
             <xs:sequence>
               <xs:element name="description" type="xs:string"
                           minOccurs="0"/>
               <xs:element name="problem" type="pluginType"/>
               <xs:element name="heuristic" type="pluginType"/>
               <xs:element name="exitCondition" type="pluginType"/>
                     <!-- outputHandler -->
               <xs:element name="outputHandler" minOccurs="0">
                 <xs:complexType>
                   <xs:sequence>
                        <!-- category-->
                     <xs:element name="category" maxOccurs="unbounded">
                       <xs:complexType>
                         <xs:simpleContent>
                           <xs:extension base="xs:string">
                             <xs:attribute name="filename"
                                           type="xs:string"/>
                             <xs:attribute name="name" type="xs:string"/>
                             <xs:attribute name="intervals"
                                           type="xs:string"/>
                           </xs:extension>
                         </xs:simpleContent>
                       </xs:complexType>
                     </xs:element>
                        <!-- End of category-->
                   </xs:sequence>
                 </xs:complexType>
               </xs:element>
                     <!-- End of "outputHandler" -->
             </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
   </xs:element>

       <!-- defining a type "pluginType" for reuse in heuristic,
            problem and exitcondition -->
   <xs:complexType name="pluginType">
```

```
      <xs:sequence>
        <xs:element name="plugin" type="xs:string"/>
        <xs:element name="class" type="xs:string"/>
        <xs:element name="parameters" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="parameter" maxOccurs="unbounded"
                          minOccurs="0">
                <xs:complexType>
                  <xs:attribute name="name" type="xs:string"
                                use="required"/>
                  <xs:attribute name="value" type="xs:string"
                                use="required"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

B The GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General

Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this

License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> one line to give the program's name and a brief idea of what it does.
> Copyright (C) yyyy name of author
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free

Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.
If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) yyyy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.
You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.