The OPT4J Documentation

December/15/2009

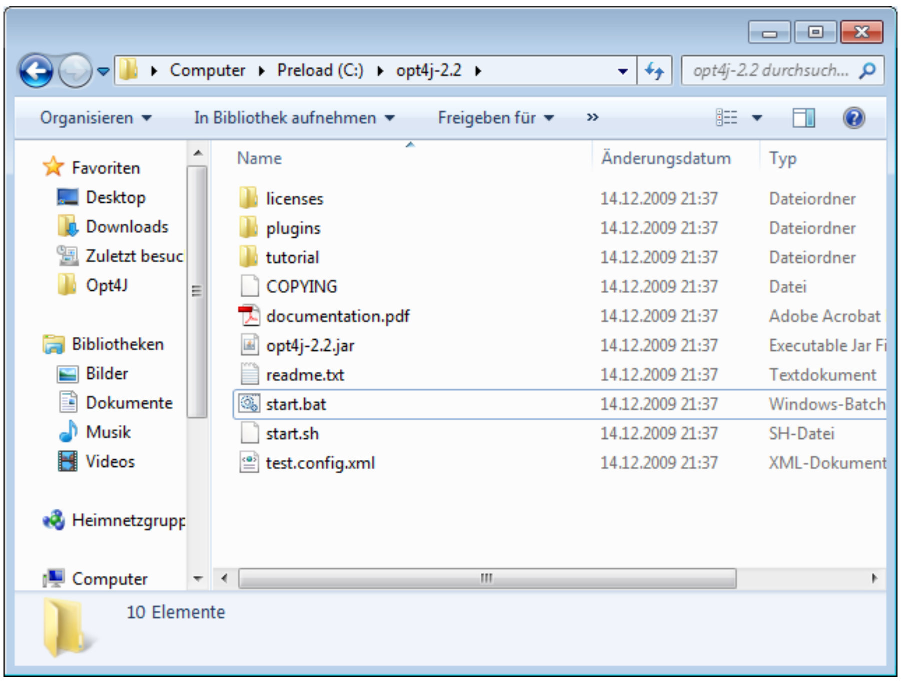
1 Introduction

This documentation consists of six sections: The first section is the introduction containing an installation guide that introduces the graphical configurator interface and the optimization viewer. The second section outlines the structure of OPT4J. A tutorial for the implementation of three problems (Hello World, Traveling Salesman, and MinOnes) and one optimizer (MutateOptimizer) are given in the following four sections. The last section shows how OPT4J can be started from command line for testcases or integrated into third party software.

1.1 Installation

To install OPT4J, the first step is to ensure that at least Java 6 Runtime Environment (JRE) is installed on your system. (You can use `java -version` on the command line (console) to check the version of your JRE.) If this is not the case, download and install the latest JRE for your operating system from http://java.sun.com.

Next, you have to download OPT4J from http://www.opt4j.org. Download the latest release (currently, the zip-file **opt4j-2.2.zip** or with additional sources **opt4j-2.2-sources.zip**). Unzip all files.



Name	Änderungsdatum	Typ
licenses	14.12.2009 21:37	Dateiordner
plugins	14.12.2009 21:37	Dateiordner
tutorial	14.12.2009 21:37	Dateiordner
COPYING	14.12.2009 21:37	Datei
documentation.pdf	14.12.2009 21:37	Adobe Acrobat
opt4j-2.2.jar	14.12.2009 21:37	Executable Jar Fi
readme.txt	14.12.2009 21:37	Textdokument
start.bat	14.12.2009 21:37	Windows-Batch
start.sh	14.12.2009 21:37	SH-Datei
test.config.xml	14.12.2009 21:37	XML-Dokument

The full release contains

- the **opt4j-2.2.jar**
- the documentation as a *pdf* file

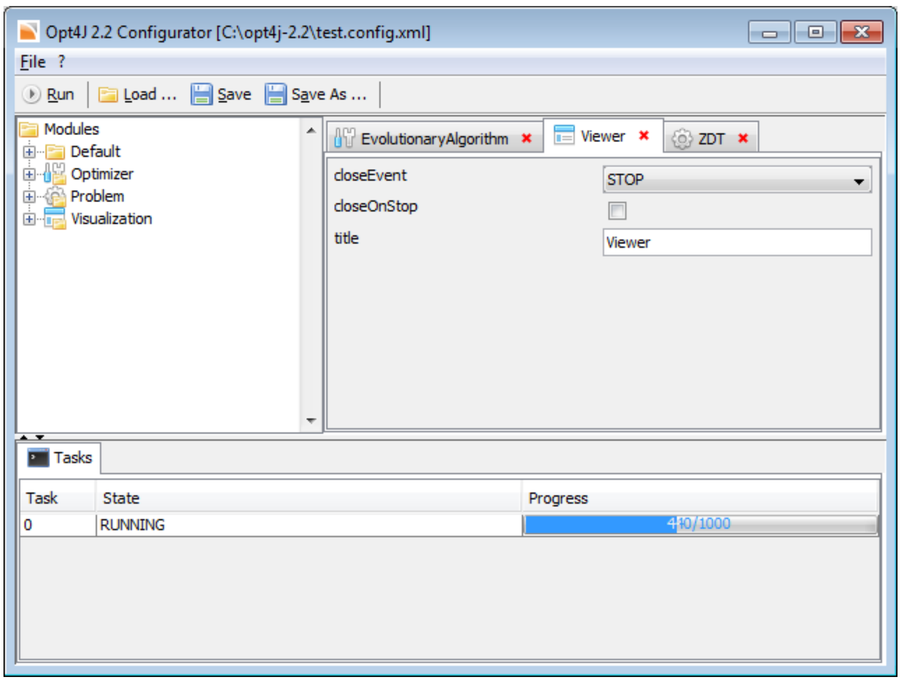
- the tutorial as sources and as a compiled *jar* in the *tutorial* folder
- the *plugins* folder from which *jar* files are automatically added to the classpath.

Use the command `java -jar opt4j-2.2.jar` to start OPT4J. Alternatively, on Windows systems, you can start the OPT4J configurator interface with the **start.bat** file (or double-click the **opt4j-2.2.jar** file). On UNIX systems, use **start.sh**.

The **opt4j-2.2-sources.zip** also contains the sources and the javadocs. Moreover, the `libsrc` folder contains an **opt4j-2.2.jar** that also includes the sources for simplified debugging.

1.2 Configurator and Viewer

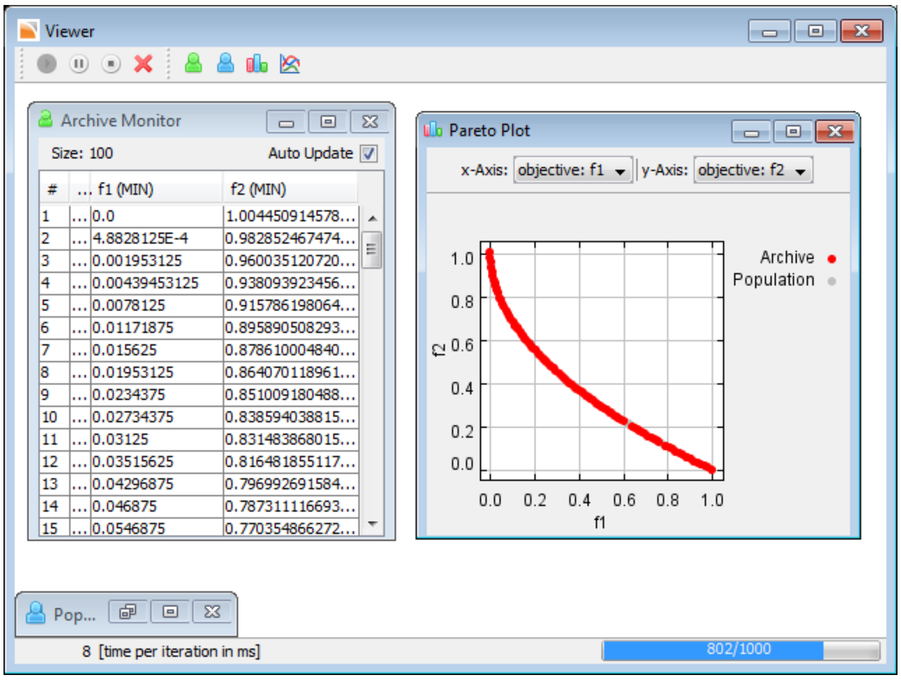
For a first test of OPT4J, you can select the following modules or you can load the **test.config.xml** file:



closeEvent	STOP
closeOnStop	☐
title	Viewer

Task	State	Progress
0	RUNNING	410/1000

Finally, click the *run* button to start the optimization:



Size: 100 Auto Update ✓

#	...	f1 (MIN)	f2 (MIN)
1	...	0.0	1.004450914578...
2	...	4.8828125E-4	0.982852467474...
3	...	0.001953125	0.960035120720...
4	...	0.00439453125	0.938093923456...
5	...	0.0078125	0.915786198064...
6	...	0.01171875	0.895890508293...
7	...	0.015625	0.878610004840...
8	...	0.01953125	0.864070118961...
9	...	0.0234375	0.851009180488...
10	...	0.02734375	0.838594038815...
11	...	0.03125	0.831483868015...
12	...	0.03515625	0.816481855117...
13	...	0.04296875	0.796992691584...
14	...	0.046875	0.787311116693...
15	...	0.0546875	0.770354866272...

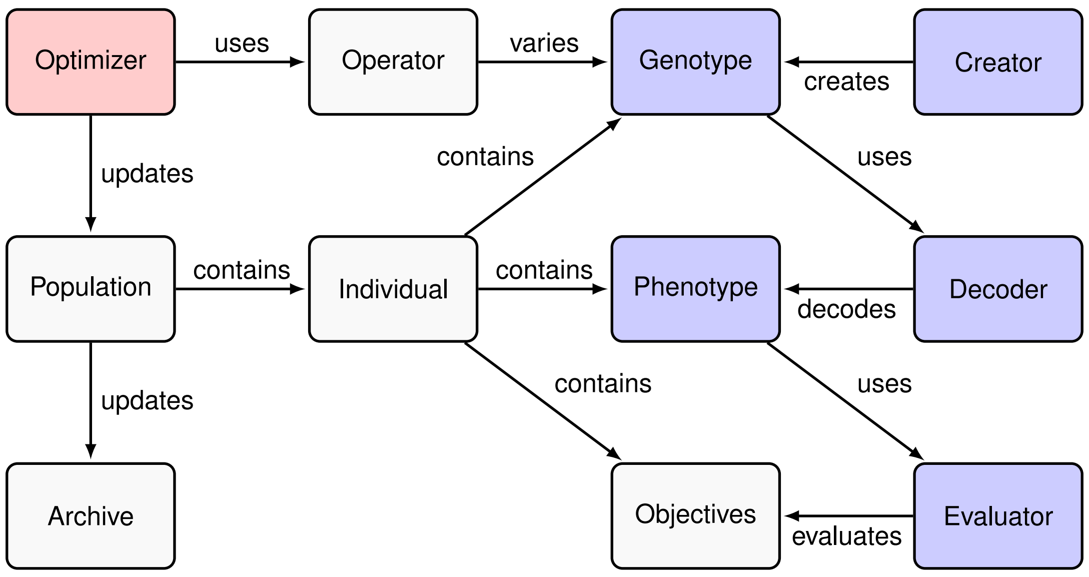
8 [time per iteration in ms] 802/1000

The OPT4J configuration files are *XML* files that define the used modules and adjusted parameters. Starting these configuration files without the configurator interfaces can be done using the command `java -jar opt4j-2.2.jar -s test.config.xml` with **test.config.xml** being the *XML* configuration. Using the `-s` parameter, you can specify multiple configurations that are executed subsequently.

2 Understanding OPT4J

2.1 The Interfaces Concept

OPT4J consists of many interfaces. By default these interfaces are implemented by predefined default classes. However, the user can implement custom implementations of these interfaces. A schematic overview of the most important OPT4J interfaces and classes can be found in the following figure:



The most important interfaces of the framework are explained in the following.

2.1.1 Optimizer (Singleton)

All *Optimizers* implement the *Optimizer* interface. Usually all *Optimizers* are derived from the *Abstract-Optimizer* that already implements some essential methods.

2.1.2 Creator, Decoder, Evaluator (Singletons)

The *Creator*, *Decoder*, and *Evaluator* interfaces are interfaces for the *Problem*. The *Creator* is responsible for the construction of random *Genotypes*, the *Decoder* converts the *Genotype* into a *Phenotype*, and the *Evaluator* calculates the *Objectives* for a *Phenotype*.

2.1.3 Genotype, Phenotype, Objectives

The *Genotype*, *Phenotype*, and *Objectives* define one *Individual*. The *Genotype* is the genetic representation of an *Individual*. There are many predefined *Genotypes* classes like for binary strings or double vectors. The *Phenotype* is the decoded representation of an *Individual*. The *Objectives* contain the results of the evaluated objective for a *Phenotype*.

2.1.4 Population (Singleton)

The *Population* is the interface and the implementation in one class. The *Population* contains all current *Individuals*.

2.1.5 Archive (Singleton)

The *Archive* interface is by default bound to the *UnboundedArchive*. The *Archive* contains the best *Individuals* that were found so far.

2.1.6 Operator: Copy, Mutate, Neighbor, Crossover (Singletons)

The *Operators* like *Copy*, *Mutate*, *Neighbor*, and *Crossover* are interfaces that are by default bound to generic types that can handle different types of *Genotypes*. *Operators* can change or create *Genotypes*.

2.1.7 Completer (Singleton)

The *Completer* is responsible for decoding and evaluating *Individuals*. By default, the *Completer* is bound to the *SequentialCompleter* that completes *Individuals* subsequently.

2.2 Using the Guice Dependency Injection

OPT4J is using the *Google Guice Dependency Injection* to wire everything together. As mentioned, the framework consists of many interfaces and the user has to set the implementations for these interfaces. For instance, the *Optimizer* interface can be implemented by the *EvolutionaryAlgorithm*. Which interface is bound to which implementation is defined in the *Modules*.

3 Tutorial (Problem I): HelloWorld

This tutorial is inspired by the *Watchmaker* framework. The goal is to vary a string of 11 characters such that the word *HELLO WORLD* is found. It introduces the basic structure of the OPT4J framework using the Creator, Decoder, and Evaluator classes as well as the Genotype and Phenotype classes.

The *Genotype* is a list of 11 characters. Here, a random *SelectGenotype* is created by the *HelloWord-Creator*:

HelloWordCreator.java

```
public class HelloWorldCreator implements Creator<SelectGenotype<Character>> {

  List<Character> chars = new ArrayList<Character>();
  {
    for (char c = 'A'; c <= 'Z'; c++) {
      chars.add(c);
    }
    chars.add(' ');
  }
  Random random = new Random();

  public SelectGenotype<Character> create() {
    SelectGenotype<Character> genotype = new SelectGenotype<Character>(chars);
    genotype.init(random, 11);
    return genotype;
  }
}
```

Since the *Phenotype* interface cannot be added to the *String* class, the *PhenotypeWrapper* is used. The *HelloWordDecoder* decodes the *Genotype* into a *Phenotype*:

HelloWordDecoder.java

```
public class HelloWorldDecoder implements Decoder<SelectGenotype<Character>,
    PhenotypeWrapper<String>> {

  public PhenotypeWrapper<String> decode(SelectGenotype<Character> genotype) {
    String s = "";
    for(int i=0; i<genotype.size(); i++){
      s += genotype.getValue(i);
    }
    return new PhenotypeWrapper<String>(s);
  }
}
```

Finally, the *Phenotype* has to be evaluated. Here, the number of matches in compared to *HELLO WORLD* are counted and maximized in the *HelloWordEvaluator*:

HelloWordEvaluator.java

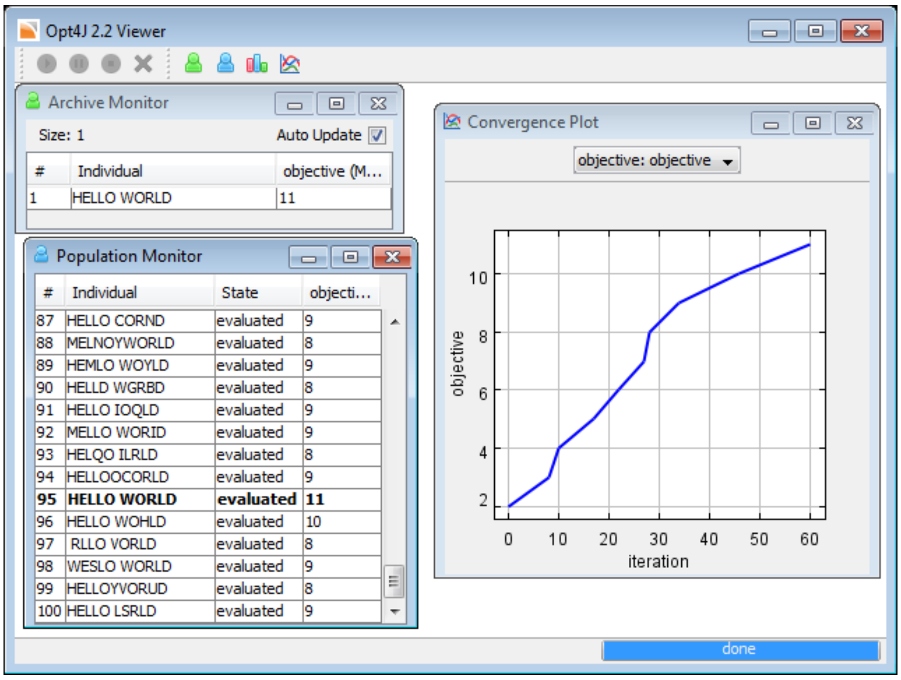
```
public class HelloWorldEvaluator implements Evaluator<PhenotypeWrapper<String>> {
  Objective objective = new Objective("objective", Sign.MAX);

  String target = "HELLO WORLD";

  public Objectives evaluate(PhenotypeWrapper<String> phenotype) {
    String s = phenotype.get();

    int value = 0;
    for (int i = 0; i < s.length(); i++) {
      if (s.charAt(i) == target.charAt(i)) {
        value++;
      }
    }

    Objectives objectives = new Objectives();
    objectives.add(objective, value);
    return objectives;
  }

  public Collection<Objective> getObjectives() {
    return Arrays.asList(objective);
  }
}
```

A *ProblemModule* for this problem is defined in the *HelloWorldModule*:

HelloWordModule.java

```
public class HelloWorldModule extends ProblemModule {

  @Override
  protected void config() {
    bindProblem(HelloWorldCreator.class, HelloWorldDecoder.class,
        HelloWorldEvaluator.class);
  }
}
```

The result looks as follows:



Size: 1 Auto Update ✓

#	Individual	objective (M…
1	HELLO WORLD	11

#	Individual	State	objecti…
87	HELLO CORND	evaluated	9
88	MELNOYWORLD	evaluated	8
89	HEMLO WOYLD	evaluated	9
90	HELLD WGRBD	evaluated	8
91	HELLO IOQLD	evaluated	9
92	MELLO WORID	evaluated	9
93	HELQO ILRLD	evaluated	8
94	HELLOOCORLD	evaluated	9
95	**HELLO WORLD**	**evaluated**	**11**
96	HELLO WOHLD	evaluated	10
97	RLLO VORLD	evaluated	8
98	WESLO WORLD	evaluated	9
99	HELLOYVORUD	evaluated	8
100	HELLO LSRLD	evaluated	9

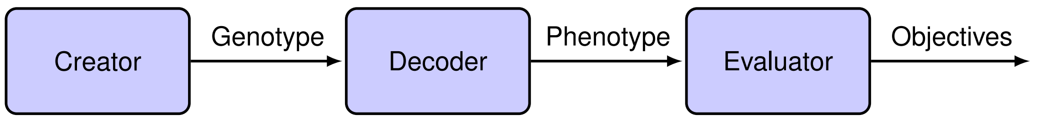
4 Tutorial (Problem II): Traveling Salesman

The Traveling Salesman Problem (http://en.wikipedia.org/wiki/Traveling_salesman_problem) is a classic single objective optimization problem. Given a set of *Cities*, the goal is to find the shortest round trip (*Route*) that contains all cities. In this tutorial, it is shown first how to implement the TSP as an *optimization problem* for OPT4J. Afterwards, a simple visualization for the TSP is presented and included in the OPT4J viewer.

4.1 Problem

To compile the following code, you have to ensure that *opt4j-2.2.jar* is on the classpath.

Put simply, an *optimization problem* is defined by the *Creator*, *Decoder*, and *Evaluator*. The *Creator* creates a new *Genotype* and passes it to the *Decoder*. The *Decoder* translates the *Genotype* into a *Phenotype*. The *Evaluator* evaluates the *Objectives*.



In the Traveling Salesman Problem, the *Genotype* is a *PermutationGenotype* of the cities representing one round trip (*Route*). The *Phenotype* is a *Route*. The *Creator*, *Decoder*, and *Evaluator* interfaces are, thus, implemented by the *SalesmanCreator*, the *SalesmanDecoder*, and the *SalesmanEvaluator*.

The *SalesmanProblem* contains the problem description. It is defined as a set of *n* cities over the area 100 times 100. The class offers a public method (`getCities()`) to retrieve these *Cities*:

SalesmanProblem.java

```java
public class SalesmanProblem {

  protected Set<City> cities = new HashSet<City>();

  public class City {
    protected final double x;
    protected final double y;

    public City(double x, double y) {
      this.x = x;
      this.y = y;
    }

    public double getX() {
      return x;
    }

    public double getY() {
      return y;
    }
  }

  @Inject
  public SalesmanProblem(@Constant(value = "size") int size) {
    Random random = new Random(0);

    for (int i = 0; i < size; i++) {
      final double x = random.nextDouble() * 100;
      final double y = random.nextDouble() * 100;
      final City city = new City(x, y);

      cities.add(city);
    }
  }

  public Set<City> getCities() {
    return cities;
  }

}
```

The number of the cities is passed in the constructor. Note that the constructor has to be annotated with `@Inject` to show the dependency injection which constructor to use. The parameter size of the constructor is annotated with `@Constant(value="size")`. We will show later how this parameter is set in a module.

The ***SalesmanCreator*** has to create *PermutationGenotypes*. Each of them contains all cities in a randomized order. The randomization is done using the `shuffle()` method of the `Collections` class:

SalesmanCreator.java

```java
public class SalesmanCreator implements Creator<PermutationGenotype<City>> {

  protected final SalesmanProblem problem;

  @Inject
  public SalesmanCreator(SalesmanProblem problem) {
    this.problem = problem;
  }

  public PermutationGenotype<City> create() {
    PermutationGenotype<City> genotype = new PermutationGenotype<City>();
    for (City city : problem.getCities()) {
      genotype.add(city);
    }

    Collections.shuffle(genotype);

    return genotype;
  }
}
```

The constructor is annotated with `@Inject`. The dependency injection will automatically construct this

Creator and pass the *SalesmanProblem* into the constructor.

Now, a conversion of the *Genotype* to the *Phenotype* has to be done by the *Decoder*. In this case, the *Phenotype* is a *Route* representing a round trip. The ***Route*** is given as an ordered *List* of *City* values.

SalesmanRoute.java

```
  public class SalesmanRoute extends ArrayList<City> implements Phenotype{}
```

Since the *SalesmanGenotype* is already given as a permutation of all cities, the ***SalesmanDecoder*** is a straightforward copy operation of the elements of the *PermutationGenotype* to the *Route*.

SalesmanDecoder.java

```
public class SalesmanDecoder implements
    Decoder<PermutationGenotype<City>, Route> {

  public Route decode(PermutationGenotype<City> genotype) {
    Route route = new Route();
    for (City city : genotype) {
      route.add(city);
    }
    return route;
  }
}
```

The next task is to evaluate the distance of a *SalesmanRoute*. The ***SalesmanEvaluator*** sums up the Euclidean *distances* between each point (*City*) of the route. The result is returned as the *Objectives* that consist of the distance *Objective* and the calculated distance *Value*. Note that the *Objectives* can contain multiple *Objective* - *Value* pairs.

SalesmanEvaluator.java

```
public class SalesmanEvaluator implements Evaluator<Route> {

  Objective distance = new Objective("distance", Sign.MIN);

  public Objectives evaluate(Route route) {
    double dist = 0;
    for (int i = 0; i < route.size(); i++) {
      City one = route.get(i);
      City two = route.get((i + 1) % route.size());
      dist += getEuclideanDistance(one, two);
    }
    Objectives objectives = new Objectives();
    objectives.add(distance, dist);

    return objectives;
  }

  private double getEuclideanDistance(City one, City two) {
    final double x = one.getX() - two.getX();
    final double y = one.getY() - two.getY();
    return Math.sqrt(x * x + y * y);
  }
  public List<Objective> getObjectives() {
    return Arrays.asList(distance);
  }
}
```

This TSP has a single objective that has to be minimized. Thus, the `distance` *Objective* is the only objective with the name `"distance"` which has to be minimized (`Sign.MIN`).

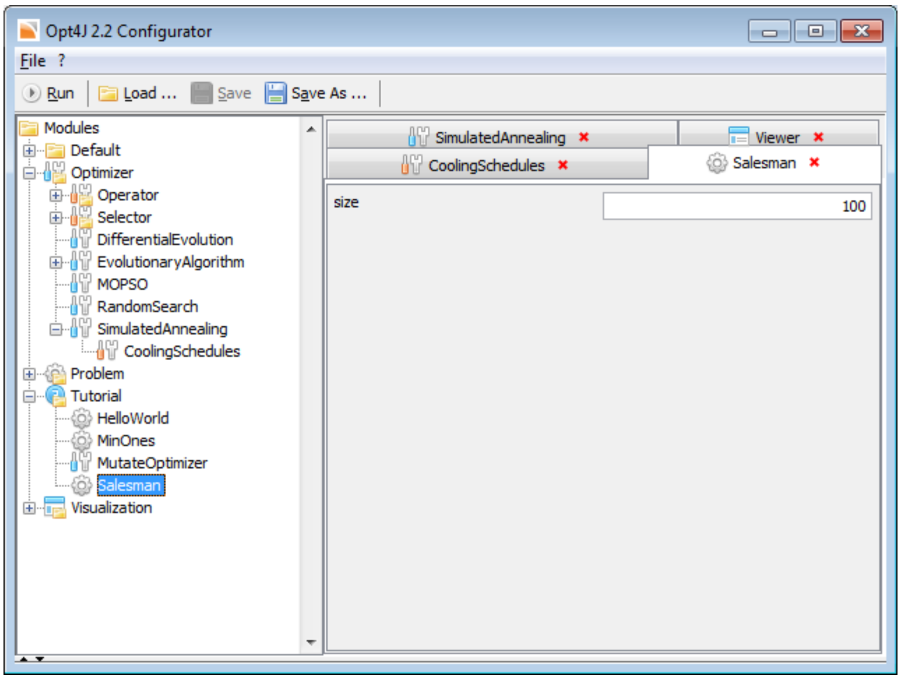
At this point the *Creator*, *Decoder*, *Evaluator*, *Genotype*, and *Phenotype* interfaces are implemented by the introduced classes for the TSP. Whats left to do is *binding* the implementations to the interfaces. Therefore, we need a so-called *Module*. Constructing and wiring the implemented classes is accomplished within the Guice framework which is integrated in OPT4J (`http://code.google.com/p/google-guice/`). The following class is also understandable without knowledge of *Guice*. The ***SalesmanModule*** is derived from the *ProblemModule* that identifies this as a configurable problem-related OPT4J module.

TravelingSalesmanModule.java

```
public class SalesmanModule extends ProblemModule {

  @Constant(value = "size")
  protected int size = 100;

  public int getSize() {
    return size;
  }

  public void setSize(int size) {
    this.size = size;
  }

  public void config() {
    bindProblem(SalesmanCreator.class, SalesmanDecoder.class,
        SalesmanEvaluator.class);
  }
}
```

The number of the cities is defined as an integer property *size*. Ensure that the property *size* has a getter and setter in the module, otherwise it will be ignored. The `@Constant(value="size")` is the same as in the constructor of the *SalesmanProblem* and, thus, this value is injected into the constructor. Note that the *Constant* annotation also allows to define a *namespace* as a class to prevent conflicts for constant values.

Now, we can start OPT4J and if the *SalesmanModule* is on the classpath it will appear in the configurator. You can simply export the classes into one jar and put this into the **plugins** folder of OPT4J. All classes in the **plugins** folder are added automatically to the classpath. You can use the following configuration to start the optimization process.



size 100

You can start the optimization and see how the route is iteratively improved.

4.2 Visualization

Loading the *Viewer* gives the user the opportunity to monitor the optimization process. In the following, this tutorial will show how you can implement a visualization of the current optimal *Route* and it to the viewer.

The ***SalesmanWidget*** shows the route graphically by double clicking the *route* in the *ArchiveViewer* or right clicking and selecting *show route*:

SalesmanWidgetService.java

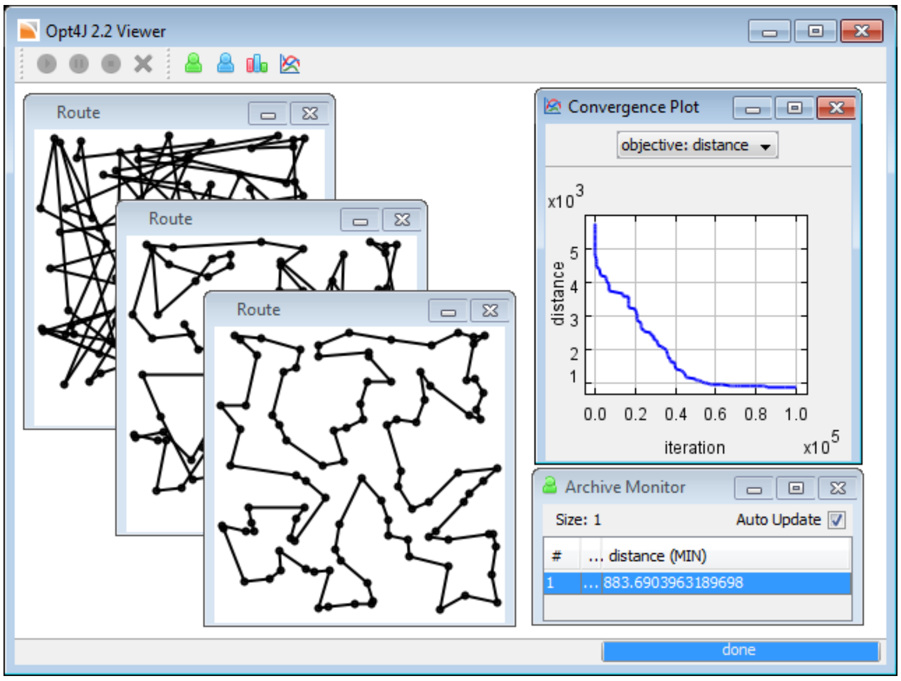
```
// The SalesmanWidgetService is an additional feature of this tutorial. It enables
     the visualization
// of a single route in the viewer.
public class SalesmanWidgetService implements IndividualMouseListener {

  // Panel that paints a single SalesmanRoute (which is the phenotype of an
  // Individual)
  public class MyPanel extends JPanel {

    protected final Individual individual;

    public MyPanel(Individual individual) {
      super();
      this.individual = individual;
      setPreferredSize(new Dimension(208, 208));
    }
    protected void paintComponent(Graphics g) {
      Graphics2D g2d = (Graphics2D) g;
      g2d.setBackground(Color.WHITE);
      g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
          RenderingHints.VALUE_ANTIALIAS_ON);
      g2d.setStroke(new BasicStroke(2f));
      g2d.clearRect(0, 0, 208, 212);

      SalesmanRoute salesmanRoute = (SalesmanRoute) individual
          .getPhenotype();

      for (int i = 0; i < salesmanRoute.size(); i++) {
        final int j = (i + 1) % salesmanRoute.size();
        City one = salesmanRoute.get(i);
        City two = salesmanRoute.get(j);

        int x1 = (int) (one.getX() * 2) + 4;
        int y1 = (int) (one.getY() * 2) + 4;
        int x2 = (int) (two.getX() * 2) + 4;
        int y2 = (int) (two.getY() * 2) + 4;

        g2d.drawLine(x1, y1, x2, y2);
        g2d.drawOval(x1 - 2, y1 - 2, 4, 4);

      }
    }
  }

  // Use a custom widget
  @WidgetParameters(title = "Route", resizable = false, maximizable = false)
  protected class SalesmanWidget implements Widget {

    final Individual individual;

    public SalesmanWidget(Individual individual) {
      super();
      this.individual = individual;
    }

    public JPanel getPanel() {
      JPanel panel = new MyPanel(individual);
      return panel;
    }

    public void init(Viewport viewport) {
    }

  }

  protected final Viewport viewport;

  // The route is shown by a double click of a individual in the archive
  // monitor panel. Thus we need the ArchiveMonitorPanel and the main
  // GUIFrame.
  @Inject
  public SalesmanWidgetService(Viewport viewport) {
    this.viewport = viewport;
  }

  // If an individual is double clicked, paint the route.
  public void onDoubleClick(Individual individual, Component table, Point p) {
    paintRoute(individual);
  }

  // If an individual is clicked with the right mouse button, open a popup
  // menu that contains the option to paint the route.
  public void onPopup(final Individual individual, Component table, Point p,
      JPopupMenu menu) {
    JMenuItem paint = new JMenuItem("show route");
    menu.add(paint);

    paint.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        paintRoute(individual);
      }
    });

  }

  // Paint the route: Construct a JInternalFrame, add the MyPanel and add the
  // frame to the desktop of the main GUIFrame.
  protected void paintRoute(Individual individual) {
    Widget widget = new SalesmanWidget(individual);
    viewport.addWidget(widget);
  }

}
```

The ***SalesmanModule*** has to be extended with a single line that tells that the *ArchiveWidget* that there is a new *IndividualMouseListener*.

SalesmanModule.java

```
public class SalesmanModule extends ProblemModule {

  @Constant(value = "size")
  protected int size = 100;

  public int getSize() {
    return size;
  }

  public void setSize(int size) {
    this.size = size;
  }

  public void config() {
    bindProblem(SalesmanCreator.class, SalesmanDecoder.class,
        SalesmanEvaluator.class);
    VisualizationModule.addIndividualMouseListener(binder(),
        SalesmanWidgetService.class);
  }
}
```

The result if an individual is double-clicked in the archive widget is the following:



Size: 1 Auto Update ✓

#	... distance (MIN)
1	... 883.6903963189698

5 Tutorial (Problem III): MinOnes Optimization of 3-SAT

Many optimization problems with a discrete search space consist of binary variables. However, in some cases not all representations of these variables are *feasible*. Common approaches deteriorate the objectives if the solution is not feasible. As a matter of fact, in case that the number of feasible solutions is much lower than the number of infeasible solutions, the optimization process is more focused on the search of feasible solutions than optimizing the objectives.

The *SAT-Decoding* package (*org.opt4j.sat*) in OPT4J includes our latest research on this topic. The specialized *Decoder* gives the user the opportunity to define constraints that have to be fulfilled to obtain a feasible solution. Thus, the Decoder takes over the task to deliver feasible solutions and, thus, allows the optimizer to deal with feasible solutions only.

As an example for the SAT-Decoding we will use the following example: Given is a vector with a fixed number of binary (*0/1*) variables. The objective is to minimize the number of *1s* of this vector. The solution of this problem is quite simple: Set all variables to *0*. However, in order to make this problem more difficult, we introduce the following condition: A solution is only considered feasible if the variables satisfy a set of *constraints*. The following example is used to illustrate this problem:

```
minimize w + x + y + z
subject to:
        w + x + y > 0 (constraint 1)
        y + z > 0     (constraint 2)
        w + z > 0     (constraint 3)
        x + y + z > 0 (constraint 4)
```

Now `(w=1,x=1,y=0,z=1)` is a feasible solution that fulfills all constraints and the objective is 3. On the other hand, for `(w=0,x=0,y=0,z=0)`, the objective is 0 but this solution is not feasible and therefore worthless. A good solution that fulfills all constraints would be `(w=0,x=1,y=0,z=1)`with the objective value 2.

In general, there is no simple algorithm that delivers feasible solutions for a set of linear constraints with binary variables since this problem is known to be *NP-complete*. However, the SAT-Decoding technique utilizes a *Pseudo-Boolean solver* to obtain feasible solutions only.

With this background knowledge, the following tutorial gives a short introduction how to use the SAT-Decoding package. First, the solution vector is created as the *Phenotype* of this problem. The variables here are integers. However, one solution is put into the **MinOnesResult.java** which is a simple map from the variables (integers) to Boolean values.

```java
public class MinOnesResult extends HashMap<Integer, Boolean> implements Phenotype
    {}
```

The ***MinOnesEvaluator*** simply counts the number of *1s*.

MinOnesEvaluator.java

```java
public class MinOnesEvaluator implements Evaluator<Result> {

  Objective ones = new Objective("ones", Sign.MIN);

  public Objectives evaluate(Result result) {

    int value = 0;
    for (boolean v : result.values()) {
      if (v) {
        value++;
      }
    }

    Objectives objectives = new Objectives();
    objectives.add(ones, value);

    return objectives;
  }

  public List<Objective> getObjectives() {
    return Arrays.asList(ones);
  }
}
```

In order to use the SAT-Decoding, we have to derive our *Decoder* from the *AbstractSATDecoder*. In fact, the *AbstractSATDecoder* is a *Decoder* and *Creator* in a single class such that we only have to specify the constraints and the rules how a feasible solution is translated into our *Phenotype*. The ***MinOnesDecoder*** is derived from the *AbstractSATDecoder* with the generic values *Genotype* (which we can ignore since we want the *AbstractSATDecoder* to handle the *Creator* task) and *MinOnesResult* which is the *Phenotype* of our problem.

MinOnesDecoder.java

```java
public class MinOnesDecoder extends AbstractSATDecoder<Genotype, MinOnesResult> {

  @Inject
  public MinOnesDecoder(SATManager manager, Rand random) {
    super(manager, random);
  }

  // Here you can set the constraints of your problem. In our case, we will
  // randomly generate a problem as a 3SAT problem (3 literals per clause)
  // with 1000 variables and 1000 clauses. This problem is known to be
  // NP-complete. However, we hope that there exists at least one feasible
  // solution (and with the seed 0 of random it does ;) ).
  @Override
  public void init(Set<Constraint> constraints) {

    Random random = new Random(0);

    for (int i = 0; i < 1000; i++) {
      Constraint clause = new Constraint(">=", 1);
      HashSet<Integer> vars = new HashSet<Integer>();
      do {
        vars.add(random.nextInt(1000));
      } while (vars.size() < 3);

      for (int n : vars) {
        clause.add(new Literal(n, random.nextBoolean()));
      }

      constraints.add(clause);
    }

  }

  @Override
  public MinOnesResult convertModel(Model model) {

    MinOnesResult minOnesResult = new MinOnesResult();

    for (int i = 0; i < 1000; i++) {
      if (model.get(i) == null || model.get(i) == false) {
        minOnesResult.put(i, false);
      } else {
        minOnesResult.put(i, true);
      }
    }

    return minOnesResult;
  }

}
```

The constructor takes the *SATManager* and *Rand* which we pass directly into the constructor of the *AbstractSATDecoder*. The `@Inject` annotation tells the dependency injection framework that this is the main constructor to use.

The constraints are set within the `init(Set<Constraint> constraints)` method. This method is overridden from the *AbstractSATDecoder* and called once when the *Decoder* is initialized. We generate a random 3-SAT (http://en.wikipedia.org/wiki/3SAT) problem with 1000 constraints from 1000 variables.

The `convertModel(Model model)` method is overridden from the *AbstractSATDecoder* and responsible for converting the feasible solutions (*Models*) into the *Phenotype* or *Result*, respectively. Note that the *Model* can contain a *null* for a variable. A *null* indicates that this is a don't care variable under the current solution and we can set this variable to *false* since we are trying to minimize the *1s*.

Finally, we have to write a *MinOnesModule* for this problem, cf. the first tutorial.

MinOnesModule.java

```
public class MinOnesModule extends ProblemModule {

  public void configure() {
    bindProblem(MinOnesDecoder.class, MinOnesDecoder.class,
        MinOnesEvaluator.class);
  }

}
```

6 Tutorial (Optimizer): Mutate Optimizer

This Tutorial shows how to write an *Optimizer* that is based on *mutation* only. The presented *Optimizer* has a population size of 100 and creates 25 offspring *Individuals* from 25 parents *Individuals* each generation by a *Mutate* operation. Afterwards, the worst *Individuals* are sorted out by a *Selector*. Fortunately, OPT4J already offers the *Mutate* operator as well as the *Selector*, making this *Optimizer* quite simple to implement.

We recommend to derive each *Optimizer* from the *AbstractOptimizer* class. Additionally to the classes which the *AbstractOptimizer* needs, our ***MutateOptimizer*** needs a *Copy* and *Mutate* operator as well as a *Selector* and the number of iterations.

MutateOptimizer.java

```
public class MutateOptimizer extends AbstractOptimizer {

  protected final Mutate<Genotype> mutate;

  protected final Copy<Genotype> copy;

  protected final Selector selector;

  protected final int iterations;

  public static final int POPSIZE = 100;

  public static final int OFFSIZE = 25;

  @Inject
  public MutateOptimizer(Population population, Archive archive,
      IndividualBuilder individualBuilder, Completer completer,
      Control control, Selector selector, Mutate<Genotype> mutate,
      Copy<Genotype> copy, @Iterations int iterations) {
    super(population, archive, individualBuilder, completer, control);
    this.mutate = mutate;
    this.copy = copy;
    this.selector = selector;
    this.iterations = iterations;
  }

  public void optimize() throws TerminationException, StopException {
    selector.init(OFFSIZE + POPSIZE);

    for (int i = 0; i < 100; i++) {
      population.add(individualBuilder.build());
    }

    nextIteration();

    for (int i = 0; i < iterations; i++) {

      Collection<Individual> parents = selector.getParents(OFFSIZE,
          population);

      for (Individual parent : parents) {
        Genotype genotype = copy.copy(parent.getGenotype());
        mutate.mutate(genotype);

        Individual child = individualBuilder.build(genotype);
        population.add(child);
      }

      completer.complete(population);

      Collection<Individual> lames = selector.getLames(OFFSIZE,
          population);
      population.removeAll(lames);

      nextIteration();

    }
  }
}
```

The constructor expects all necessary objects for the *AbstractOptimizer* and our *Optimizer*. Our *MutateOptimizer* needs the *Copy* and *Mutate* operator, the *Selector*, and the number of *iterations*. The `int iterations` is annotated with the predefined annotation for iterations (`@Iterations`).

The optimization is done in the `optimize()` method. The Selector is initialized with the maximum size of 125. Initially, the *Population* is filled with 100 randomly generated *Individuals*. After creating this initial *Population*, we can indicate that one iteration has passed by calling the method `nextIteration()` implemented by the *AbstractOptimizer*. The method `nextIteration()` automatically evaluates all *Individuals* and updates the *Archive*.

The following optimization process takes place in the for-loop. The *Selector* selects the parent *Individuals*. The *Genotype* of each *Individual* is copied and mutated. A new offspring *Individual* is built by the *IndividualBuilder* and added to the *Population*. Now, we call the *Completer* `complete()` method to evaluate all *Individuals*. The current size of the *Population* is 125, thus, 25 low quality *Individuals* have to be removed from the *Population*. This is done by the *getLames()* method of the *Selector*. Finally, the method `nextIteration()` from the *AbstractOptimizer* has to be called for each completed iteration.

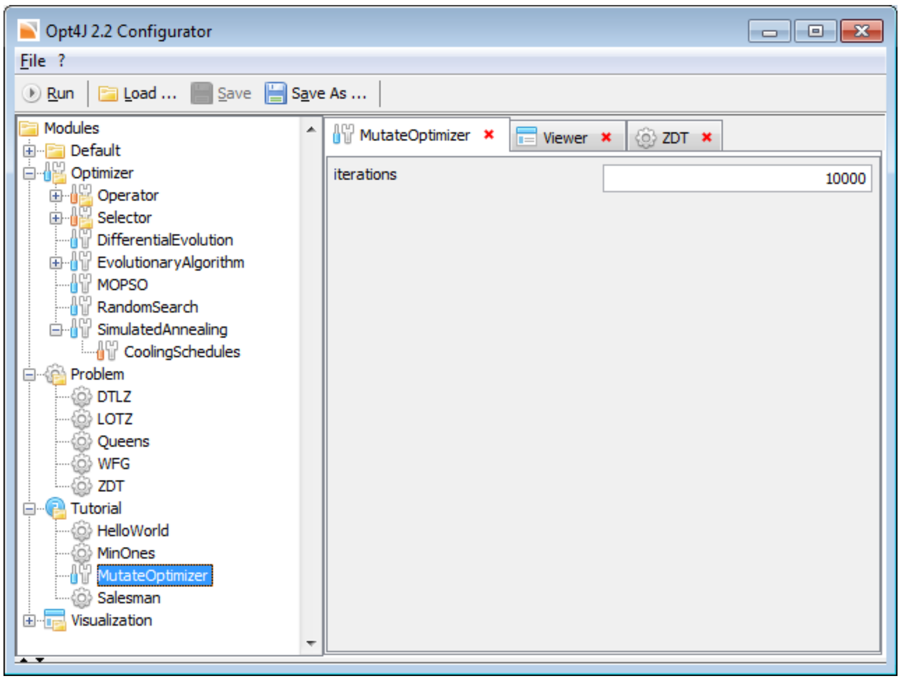
Next, we show how to write a *Module* for the *MutateOptimizer*. The ***MutateOptimizerModule*** has to be derived from the *OptimizerModule*.

MutateOptimizerModule.java

```
public class MutateOptimizerModule extends OptimizerModule {

  @Iterations
  protected int iterations = 1000;

  public int getIterations() {
    return iterations;
  }

  public void setIterations(int iterations) {
    this.iterations = iterations;
  }
  public void config() {
    bindOptimizer(MutateOptimizer.class);
  }

}
```

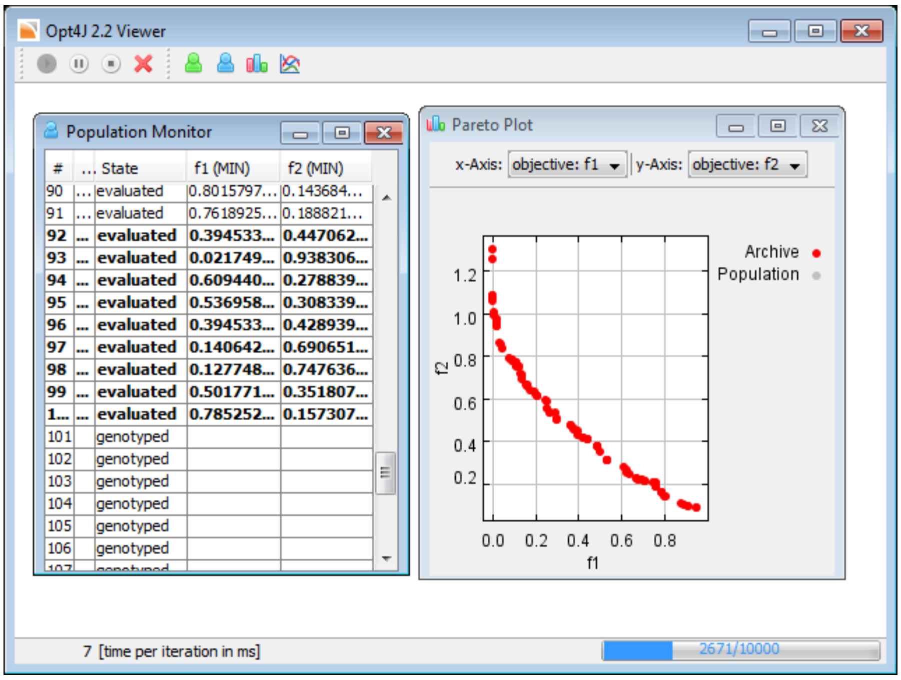
The *Optimizer* interface is bound to our *MutateOptimizer*. The iterations are wired as in the TSP tutorial.

Now, we can test our new *MutateOptimizerModule*:



iterations 10000

And after starting the optimization:



#	…	State	f1 (MIN)	f2 (MIN)
90	…	evaluated	0.8015797…	0.143684…
91	…	evaluated	0.7618925…	0.188821…
92	**…**	**evaluated**	**0.394533…**	**0.447062…**
93	**…**	**evaluated**	**0.021749…**	**0.938306…**
94	**…**	**evaluated**	**0.609440…**	**0.278839…**
95	**…**	**evaluated**	**0.536958…**	**0.308339…**
96	**…**	**evaluated**	**0.394533…**	**0.428939…**
97	**…**	**evaluated**	**0.140642…**	**0.690651…**
98	**…**	**evaluated**	**0.127748…**	**0.747636…**
99	**…**	**evaluated**	**0.501771…**	**0.351807…**
1…	**…**	**evaluated**	**0.785252…**	**0.157307…**
101		genotyped		
102		genotyped		
103		genotyped		
104		genotyped		
105		genotyped		
106		genotyped		

7 [time per iteration in ms]

2671/10000

7 Integration

This section describes how OPT4J can be embedded into third party software by starting it from the command line or directly from Java.

7.1 Start Optimization from Command Line

To start an optimization without the configurator directly from the command you need a valid configuration XML file. A configuration file can be created with the configurator by saving a configuration. In order to start a configuration `config.xml` directly from the command line, you simply you call one of the following commands:

- `java -jar opt4j-2.2.jar -s config.xml` or
- `java -cp opt4j-2.2.jar org.opt4j.start.Opt4JStarter config.xml`.

In contrast to the latter case, in the first case the splash screen will appear for a few milliseconds.

7.2 Start Optimization from Java

The following code snippet shows how to optimize the first DTLZ test function with an Evolutionary Algorithm from Java and, finally, how to obtain the best found solutions:

```java
EvolutionaryAlgorithmModule ea = new EvolutionaryAlgorithmModule();
ea.setGenerations(500);
ea.setAlpha(100);

DTLZModule dtlz = new DTLZModule();
dtlz.setFunction(DTLZModule.Function.DTLZ1);

GUIModule gui = new GUIModule();
gui.setCloseOnStop(true);

Collection<Module> modules = new ArrayList<Module>();
modules.add(ea);
modules.add(dtlz);
modules.add(gui);

Opt4JTask task = new Opt4JTask(false);
task.init(modules);

try {
  task.execute();
  Archive archive = task.getInstance(Archive.class);

  for(Individual individual: archive){
    //...
  }
```

```
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      task.close();
    }
```

First, you have to fill a collection with the desired modules for the optimization. The `Opt4JTask` is constructed with the parameter `false` that indicates that the task is closed manually and not automatically once the optimization stops. The `Opt4JTask` is initialized with the modules. Now, it is possible to execute the task (which is blocking in this case, but you can also start it in a separate thread) and once the optimization is finished you can obtain the `Archive` to iterate over the best solutions. After closing the task, obtaining instances like the `Archive` is not possible anymore.