# Tutorial Series on Java Framework for Evolutionary Computation and Decision Making

## Tutorial 4: Decision Support module

Michał K. Tomczyk

*Institute of Computing Science, Poznan University of Technology, Poznań, Poland*

This document is part of a tutorial series on JECDM. URL: www.jecdm.cs.put.poznan.pl, email: jecdm@cs.put.poznan.pl).
The framework version used when preparing the document: 1.0 (initial release; Java v. 17.0; IntelliJ IDEA 2023.2).

**Abstract**

This tutorial discusses the Decision Support module. It showcases the module's general capabilities and provides concrete examples of using it in practice.

*Keywords:* JECDM, decision support, multi-criteria decision analysis

**Contents**

**1 Introduction** **1**

**2 Understanding fundamental concepts** **2**
2.1 Overview of major components . . . . . 2
2.2 Criteria and Alternative class . . . . . 3
2.3 AbstractAlternatives class . . . . . 3
2.4 Decision-making context [T] . . . . . 3
2.5 Report . . . . . 5
2.6 Preference information [T] . . . . . 6
2.7 History of preference elicitation [T] . . . . . 8
2.8 Preference model [T] . . . . . 9

**3 Executing preference elicitation** **13**
3.1 Triggering preference elicitation [T] . . . . . 15
3.1.1 Rule for deciding when to interact . . . 15
3.1.2 Interaction trigger . . . . . 19
3.2 Analyzing and refining the alternatives [T] . . . 19
3.2.1 Termination filter . . . . . 20
3.2.2 Reduction filter . . . . . 21
3.2.3 Refiner . . . . . 21
3.3 Constructing reference sets [T] . . . . . 22
3.3.1 Reference set constructor interface . . . 23
3.3.2 Reference sets constructor class . . . . 25
3.4 Generating a feedback [T] . . . . . 26
3.5 Complex example [T] . . . . . 29

**4 Updating preference models** **30**
4.1 Constructing compatible model instances [T] . 36
4.2 Handling inconsistencies [T] . . . . . 39
4.3 Model system [T] . . . . . 40

**5 Decision Support System** **40**
5.1 Example use case #1 [T] . . . . . 42
5.2 Example use case #2 [T] . . . . . 43

**6 Implemented hybrid algorithms** **45**
6.1 IEMO/D [T] . . . . . 46
6.2 NEMO-0 and NEMO-II [T] . . . . . 47
6.3 CDEMO and DCEMO [T] . . . . . 49
6.4 Comparative analysis [T] . . . . . 50

## 1. Introduction

This document showcases how to utilize the JECDM's *DecisionSupport* module. It was designed to allow for establishing multi-criteria decision support processes and integrating them with the methods for multi-objective optimization. Note that this document assumes the reader is already familiar with the multi-criteria decision analysis field [1], especially with such concepts as preference modeling, preference formulating, preference learning, inconsistency handling, etc.

Before proceeding, let me highlight what the user may expect from this module. First, it is worth underlining that multi-criteria decision analysis is not a particularly convenient field when it comes to developing a computational framework (not confused with developing ready-to-use software). There are two reasons for this. First, decision-reaching is not a rigorously specified process. It differs from the expert following the given steps precisely to elaborate on a final recommendation to the decision maker. On the contrary, it is instead an elastic process that, although it should follow some general guidelines, can be altered online on demand. This conflicts with designing a generalized framework that, inherently, must implement some rather strict procedures. Second, various components of this field (methods, preference models, or forms of preference information) are often mathematically incompatible. For instance,

*Email address:* michal.tomczyk@cs.put.poznan.pl (Michał K. Tomczyk)
*URL:* www.cs.put.poznan.pl/mtomczyk (Michał K. Tomczyk)

three general classes of preference models are distinguished in this area: value models, relational models, and models based on decision rules. They assess the relevance of the involved alternatives in entirely different ways. For example, value models perform a straightforward alternative → global score mapping, while the relative models base the assessment on comparing alternatives with one another. This complicates defining common and generalized interfaces/abstract classes, etc.

Due to the above reasons, the initial release of JECDM is primarily focused on establishing a fundamental architecture for simulating the decision-reaching process with the potential of being extended in the future. This also means that the goal was not to implement – and hard-code – all the methods available in the literature, and the user will currently find none of these. There is plenty of available software that provides ready-to-use decision-aiding methods. Moreover, the primary focus is currently on developing and testing the methods in highly synthetical environments (the human decision makers are simulated with artificial ones, the decision-reaching process follows precise steps, etc.). Nonetheless, the implemented architecture has great potential to be implemented in concrete software intended to be used in practice. Further, the available implementations currently concern the value models (a preference model) and pairwise comparisons (a form of preference information). Nonetheless, the code is flexible, thus open for other preference representations in future updates. Lastly, JECDM is focused on creating hybrid evolutionary methods driven by the decision maker's preferences. For this reason, the module is implemented in a way that allows coupling it with the *EvolutionaryComputation* module relatively straightforwardly.

**Important note:** This module makes the same assumptions regarding standardization as the module for evolutionary computation. Specifically, it assumes that all rescaling-depended processes will involve the data normalization, which will translate the input data points into $[0, 1]^M$ hypercube, where $\mathbf{0}$-vector is the utopia point, $\mathbf{1}$ is the nadir, and $M$ is the number of dimensions/objectives/criteria. Further, it is assumed that the alternatives' evaluations are stored as their performance vector, which means that this vector is interpreted as a criteria evaluation (or objective function value, in the context of coupling the system with a method for evolutionary multi-objective optimization).

## 2. Understanding fundamental concepts

This section examines some selected components whose introduction now may clarify further discussion. Note that some of them are presented without any specific context. Their applicability will be addressed from the broader perspective when moving to the two most essential parts of this document: discussing the preference elicitation module (Section 3) and the module for executing the preference learning (Section 4).

### *2.1. Overview of major components*

This section briefly overviews the central module's components and showcases their hierarchy. They are presented in the diagram depicted in Figure 1. At the top, there is a *DecisionSupportSystem* class that – as expected – encapsulates all the data and processes associated with the general multi-criteria decision analysis task. Below, a *DecisionMakerSystem* is showcased. It handles all the decision-maker-related data and processes (e.g., stores the decision maker's history of preference elicitation). Note that the decision-reaching process may involve more than one decision maker in reality (group decision-making). Thus, the decision support system can handle multiple decision maker systems.

Next, a *ModelSystem* is listed. As the name suggests, the model system provides preference-model-related functionalities, e.g., handles the preference model itself, provides a means for learning how to parameterize the model, and gives ways of handling inconsistencies. Note that the class type is generic – all the model-related components are implemented as generics to introduce higher-level generalizations. The framework allows coupling more than one model system with one decision maker system. This may be particularly convenient when the decision maker's best-suited model is unknown (a model that can adequately represent the decision maker's value system). By employing multiple different model systems, the system may explore various possibilities for representing the preferences simultaneously, developing more robust and credible conclusions this way.

A single model system is coupled with a single implementation of the *IPreferenceModel* interface. At this level, it serves as a model definition/signature/header (i.e., it does not tell anything about the model's parameterization). Then, the preference model can be coupled with its concrete (multiple) instances: the *AbstractPreferenceModel* class. Allowing to maintain multiple parameterizations of the preference model is in the spirit of the preference dissagregation [2] (preference learning). Note that the model definition and its instances must be consistent – by type – with their model system. However, different types of model systems may be utilized in one decision maker system.

The central components presented above introduce various functionalities that – when coupled – may simulate the multiple-criteria decision analysis process. In order to simplify the implementations, i.e., take off the burden from the programmer, these procedures were grouped into two semantically consistent modules – see Figure 2:

1. module dedicated to executing the preference elicitation,
2. module responsible for updating, i.e., learning the suitable parameterizations of the employed preference models.

These modules take control over the execution of various systems' procedures. This includes ensuring the correctness and safety of the processing flow, generating the reports on the execution, creating the results, etc. They will be covered in the following sections. Note that the future development plans include adding functionalities related to group decision-making (and creating a separate module).

**Important note:** The parameterization of the *DecisionSupportSystem* should be done via its params container. This top-level implementation hides, e.g., the creation of the modules and their explicit execution. Nonetheless, the following sections will delve into the lower-level components. Example use cases will be showcased to give the reader a good understanding

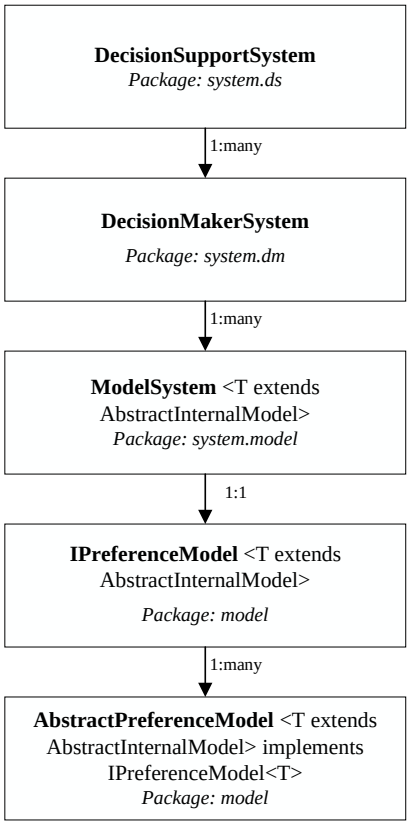



Figure 1: Hierarchy of the major components

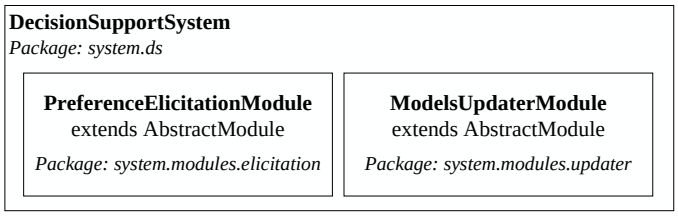



Figure 2: Decision support system modules

of the system's components. Nonetheless, focusing on them individually requires mimicking the flow of the higher-level components, which implies that they will be used in a way that omits the – very general – application programming interference. Consequently, the codes presented when discussing the *PreferenceElicitationModule* (Section 3) and the *ModelsUpdaterModule* (Section 4) illustrate the general ideas, but the system should not be ultimately configured in a way presented in these codes. The correct way of configuring the *DecisionSupportSystem* will be presented later in Section 5.

### 2.2. Criteria and Alternative class

The concept of alternative and criterion are the fundamental bricks in multi-criteria decision analysis. However, their classes were already been discussed in the tutorial document "*Tutorial 2: Evolutionary Computation module – part 1: Basics and single-objective optimization*". It is up to the reader to revisit the respective classes (*alternative* and *criterion* packages).

### 2.3. AbstractAlternatives class

This module introduces an additional abstract and generic class that serves as a simple wrapper for an array (*ArrayList*) of objects that support retrieving stored instances of the *Alternative* class. This class was introduced for two reasons. First, it makes the wrapped array read-only. Nonetheless, it is worth remembering that the stored objects that can be retrieved may not be immutable (depending on the concrete class used as a generic type; *Alternative* is not immutable itself). Overall, the class was introduced to avoid accidental modifications of the contained array. Second, the class may also serve as a bridge between various framework modules. For instance, the *Specimens* class in the population package of the *EvolutionaryComputation* module extends *AbstractAlternatives*. The type used is the *Specimens* class itself, which stores and provides a means for retrieving the alternative. Thus, the instance *Specimens* class may be passed to the decision support module, which will see it as an array of alternatives. The interested reader is referred to the *AbstractAlternatives* class and the *IAlternativeWrapper* interface in the alternative package to get acquainted more with these sources.

### 2.4. Decision-making context [T]

Source package: *t1_t10.t4_decision_support_module.* ***t1_concepts.t1_decision_making_context***

The *DMContext* (context; see the *dmcontext* package) is a class that bridges the decision-making system and the external caller, e.g., the evolutionary method. The classic decision-reaching process (i.e., not involving the optimization) operates in a quite sterile environment where all the alternatives are known in advance (usually). In turn, when implemented within an optimization process, the environment may constantly change. The context is intended to capture these changes, allowing the decision-making system to understand what is happening.

In the initial release, the context is a simple data container that passes data between the evolutionary method and the system. The most fundamental piece of data maintained is the alternative set (*AbstractAlternatives*), which – from this perspective – may be the optimizer's current population (*Specimens* class used to wrapped the array of specimens stored in *SpecimenContainer*). Note that, however, the *DecisionSupportSystem* class (will be discussed later in Section 5) is to be supplied with the contexts' params container instead of the actual object instance. The system itself will be responsible for suitably creating the context object.

Listing 1 brings the context's params container for convenience. Its fields are relatively easy to understand. First, the "*_currentIteration*" parameter indicates the current iteration number. It is up to the higher-level component to impose some interpretability on this simple counter (and ensure the consistency of subsequent calls). For instance, the iteration can mean generation when coupling the decision support system with the evolutionary algorithm. Next, the "*_currentAlternativesSupperset*" is an object that wraps all the

alternatives considered for processing. Similarly, in the context of evolutionary computation, this object can wrap the current population.

The remaining three fields, i.e., "*_currentOS*", "*_normalizationBuilder*, and "*_osChanged*", play a similar role when used in the module for evolutionary computation. First, "*_currentOS*" provides current data on the objective/performance space. Further, the context can provide the "*_normalizationBuilder*" object to dictate how the objective space data should be translated into the normalization objects. Lastly, the "*_osChanged*" flag can be suitably set to inform the system whether the data on the known objective space has changed since the previous communication with the system.

As already explained, the decision-support module follows the same standardization in normalization as the module for evolutionary computation. This means that all the calculations that inherently involve rescaling will operate on $[0, 1]^M$ hypercube, where $\mathbf{0}$-vector is the utopia point, while $\mathbf{1}$-vector is the nadir (and $M$ is the number of objectives). The rescaling is fundamental in this module when, e.g., preprocessing input alternatives during the preference elicitation phase or when learning how to parameterize employed preference models suitably. Note that the context creates the proper normalization objects when being instantiated. The constructed functions are stored as a field and are accessible.

Listing 1: DMContext class (params container

```
/**
 * Params container.
 */
public static class Params
{
  /**
   * Field representing current iteration (
       e.g., generation).
   */
  public Integer _currentIteration = null;

  /**
   * This field is supposed to contain all
       the alternatives maintained at some
       specific processing stage.
   */
  public AbstractAlternatives<?>
      _currentAlternativesSuperset;

  /**
   * This field is supposed to store the
       data on the current objective space (
       can be null).
   */
  public ObjectiveSpace _currentOS;

  /**
   * This flag indicates whether the OS
       changed from one iteration to another
       .
   */
  public boolean _osChanged;

  /**
   * Normalization builder that will be
       used to generate normalization
       objects using the OSs (if null,
   * the standard linear builder will be
       used ({@link StandardLinearBuilder}).
  */
  public INormalizationBuilder
      _normalizationBuilder;



  /**
   * Random number generator (should be
       provided when some random-based
       components are involved; they will
       try using this object).
   */
  IRandom _R;
}
```

The tutorial's source code is brought in Listing 2 for convenience. It shows how to adjust the context's params container suitably. The code is relatively straightforward and introduces nothing really new. Its last lines are devoted to creating the object instance (it is typically a prerogative of the decision support system; the criteria and the current timestamp must be supplied along the params container to instantiate the context manually) and printing the context's string representation, which may be convenient for, e.g., debugging or generating system logs. Figure 3 illustrates the expected console output. Obviously, the printed timestamps depend on when the code was executed.

Listing 2: Tutorial's source code

```
// Create object instance:
DMContext.Params pDMC = new DMContext.
    Params();
// Specify current iteration (e.g.,
    generation):
pDMC._currentIteration = 0;
// Specify the alternatives: A0 = [0.0,
    0.0] and A1 = [1.0, 1.0] (performance
    vectors)
pDMC._currentAlternativesSuperset = new
    Alternatives(
Alternative.getAlternativeArray("A",
new double[][]{{0.0d, 0.0d}, {1.0d, 1.0d}})
    );
// Specify current OS (e.g., [0, 1]^2
    bounds with the objectives to be
    minimized):
pDMC._currentOS = new ObjectiveSpace(Range.
    getDefaultRanges(2), new boolean[2]);
// Specify that the OS has not changed
    since the previous call:
pDMC._osChanged = false;
// Specify the normalization builder:
pDMC._normalizationBuilder = new
    StandardLinearBuilder();
```

```
// Typically  created by the
    DecisionSupportSystem object (sets the
    criteria and the current local date and
    time):
DMContext context = new DMContext(pDMC,
    Criteria.constructCriteria("C", 2, false
    ), LocalDateTime.now());

// Print the basic data about the context:
System.out.println(context);
```

```
Current iteration = 0
Current date and time = [7.11.2024 18:9:32:687]
System starting date and time = [7.11.2024 18:9:32:686]
No. alternatives = 2
Criteria = C0, C1
Objective space data:
Utopia point = 0,0000 0,0000
Nadir point = 1,0000 1,0000
Ranges data:
Left = 0,0000; Right = 1,0000
Left = 0,0000; Right = 1,0000
Criteria types:
Minimization
Minimization
Objective space changed = false
Normalization objects:
Linear normalization: min = 0,000000; max = 1,000000
Linear normalization: min = 0,000000; max = 1,000000
Random number generator provided = false
```

Figure 3: Expected console output

### 2.5. *Report*

It is worth highlighting that the return types of most of the relevant system components are report-like classes. They extend the common *AbstractReport* class (see the *system.modules* package). The concrete system's components introduce dedicated extensions of this class that are supposed to capture the results and provide additional data summarizing the executed processing. On the one hand, the system can suitably interpret the stored data to steer the ongoing processes. On the other hand, the class provides methods for creating string representations, which introduces practical means for generating complex logs on the decision-reaching process. Listing 3 showcases the – relatively straightforward – *AbstractReport* class for the reader's convenience.

Listing 3: AbstractReport class

```
/**
 * Abstract report-like class. Provides
    common fields and functionalities.
 *
 * @author MTomczyk
 */
public abstract class AbstractReport
{
  /**
   * Indicates the processing time (in ms).
   */
  public long _processingTime;

  /**
   * Timestamp (iteration).
   */
  public int _iteration;

  /**
   * Timestamp (date and time associated
      with the decision-making context).
   */
  public LocalDateTime _dmContextDateTime;

  /**
   * Timestamp (date and time associated
      with the object creation).
   */
  public LocalDateTime
      _reportCreationDateTime;

  /**
   * Parameterized constructor.
   *
   * @param dmContext current decision-
      making context
   */
  protected AbstractReport(DMContext
      dmContext)
  {
    _iteration = dmContext.
        getCurrentIteration();
    _dmContextDateTime = dmContext.
        getCurrentDateTime();
    _reportCreationDateTime = LocalDateTime
        .now();
  }

  [...]

  /**
   * Auxiliary method for printing the
      report.
   */
  public void printStringRepresentation()
  {
    String [] lines =
        getStringRepresentation();
    if (lines != null) for (String l: lines
        ) System.out.println(l);
  }

  [...]
}
```

### 2.6. *Preference information [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t1_concepts.t2_preference_information***

Preference information is a fundamental concept when understanding the decision maker's preferences. It is a piece of information that they can supply to allow the algorithm to understand how they evaluate the alternatives. Unfortunately, preference information is a concept that is difficult to generalize from the programming perspective. For instance, the decision maker may wish to provide the preferences directly – which is about parameterizing the employed preference model, e.g., providing weights for criteria. In turn, the decision maker may be asked to judge the presented alternatives holistically (indirectly), e.g., compare them pairwise or assign them to predefined preference classes (e.g., "highly preferred"). In this view, preference forms can be unique in their role in preference learning; thus, the implemented processing may be difficult to generalize.

Since the primary focus of the initial release of JECDM is pairwise comparisons, the programming decisions made were as follows:

- Introduce basic, very general interfaces/abstract classes with the potential to be extended to simulate new forms of preference information that may be considered in future releases.
- Create a dedicated class for pairwise comparisons as the central focus of this release.

Listing 4 brings the top-level representation of preference information: *IPreferenceInformation* interface (see the *preference.IPreferenceInformation*). Clearly, the current version of the interface is very general, minimizing the risk of API violation in future releases. For instance, "*isIndirect()*" provides the answer yes/no whether the actual instance represents an indirect preference form or not; the "*getForm()*" implementation should return the form's unique identifier (see the *preference.Form* enum); while the "*getPairwiseComparisons()*" should return the preference information viewed as an array of pairwise comparisons (null; if the translation is not possible; implementation-dependent).

Listing 4: IPreferenceInformation interface

```
/**
 * General interface for classes
    responsible for simulating the decision
     maker's preference examples
 * (preference information provided).
 *
 * @author MTomczyk
 */
public interface IPreferenceInformation
{
  /**
   * Method for returning the preference
       information viewed as pairwise
       comparisons. Should return null if
       the specific
   * form cannot be translated into this
       form.
   *
   * @return preference example viewed as a
        series of pairwise comparisons (null
        if not possible)
   */
  PairwiseComparison[]
      getPairwiseComparisons();

  /**
   * The implemented method should return
       false if the preference example is
       direct (explicit, e.g., provided
       weights),
   * true otherwise (indirect, holistic
       form, e.g., pairwise comparison).
   *
   * @return false if the preference
       example is direct, true otherwise (
       holistic)
   */
  boolean isIndirect();

  /**
   * To string method should be overwritten
       .
   *
   * @return string representation
   */
  String toString();

  /**
   * To string method should be overwritten
        (returns the string representation).
   *
   * @return string representation
   */
  String getStringRepresentation();

  /**
   * Returns the constant associated with
       the form of the preference
       informatyko represented by the object
        instance.
   * @return form.
   */
  Form getForm();

  /**
   * Supposed to return a copy of the
       object. The copy should be deep, at
       least at this object's level (i.e.,
       the
   * implementation does not need to
       guarantee that the bottom objects --
       e.g., alternatives -- will be cloned)
       .
     *
   * @return copy of the object
   */
      IPreferenceInformation getClone();
```

```
}
```

Lower than *IPreferenceInformation* in the hierarchy is the *AbstractPreferenceInformation* class that provides standard fields and functionalities (*preference* package; implements the *IPreferenceInformation* interface). The existence of two general types of preference forms, i.e., indirect and direct, motivated the introduction of an additional layer in the class hierarchy. Currently, it has only one implementation: *AbstractIndirectForm* class, dedicated to holistic forms of preference judgments (*preference.indirect* package). Finally, a dedicated *PairwiseComparison* class is introduced (extends *AbstractIndirectForm*, the same package). The overall class hierarchy leading from *IPreferenceInformation* to *PairwiseComparison* is illustrated in Figure 4.

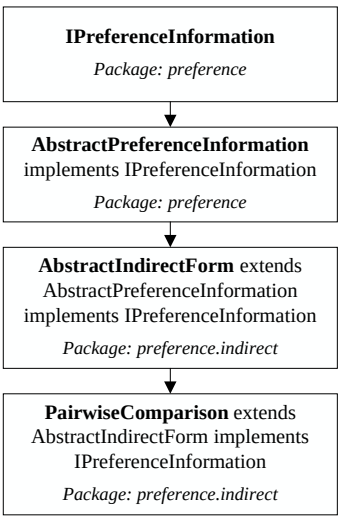



Figure 4: Class hierarchy (preference information)

Listing 5 brings the most relevant code pieces of the *PairwiseComparison* for convenience. The code is relatively straightforward and, thus, does not require elaboration. Note that the compared pair of alternatives can be assigned a relation (depending on the decision maker's indication; see the *relation.Relation* enum). For instance, the preference relation may be assigner (see the dedicated static object constructor "*getPreference*". It implies that one of the input alternatives is preferred by the decision maker. Further, the decision maker may state "indifferent" or "incomparable". Note that the current release of JECDM is focused only on the "preference" case.

Listing 5: PairwiseComparison class

```
/**
 * Class representing a pairwise comparison
    .
 *
 * @author MTomczyk
 */
public class PairwiseComparison extends
    AbstractIndirectForm implements
    IPreferenceInformation
{
  /**
   * Parameterized constructor.
   *
   * @param first    the first alternative
   * @param second   the second alternative
   * @param relation relation between the
       alternatives
   */
  protected PairwiseComparison(Alternative
      first, Alternative second, Relations
      relation)
  {
    super("Pairwise comparison: " + first.
        getName() + " " + relation.name() +
        " " + second.getName(), relation);
    _alternatives = new Alternative[]{first
        , second};
  }

  /**
   * Object constructor for the case A
       preferred over B.
   *
   * @param preferred    the preferred
       alternative
   * @param notPreferred the not preferred
       alternative
   * @return object instance
   */
  public static PairwiseComparison
      getPreference(Alternative preferred,
      Alternative notPreferred)
  {
    return new PairwiseComparison(preferred
        , notPreferred, Relations.PREFERENCE
        );
  }

  /**
   * Object constructor for the case A
       indifferent to B.
   *
   * @param A one of the indifferent
       alternatives
   * @param B the other indifferent
       alternative
   * @return object instance
   */
  public static PairwiseComparison
      getIndifference(Alternative A,
      Alternative B)
  {
    return new PairwiseComparison(A, B,
        Relations.INDIFFERENCE);
  }

  [...]

  /**
   * The method returns the preferred
       alternative. If relation is not
```

```
        PREFERENCE, the method returns null.
     *
     * @return the preferred alternative (or
        null, if they are indifferent)
    */
    public Alternative
        getPreferredAlternative()
    {
      if (_relations[0].equals(Relations.
          PREFERENCE)) return _alternatives
          [0];
      else return null;
    }

    [...]

    /**
     * Returns the constant associated with
        the form of the preference
        information represented by the object
         instance.
     *
     * @return form.
     */
    @Override
    public Form getForm()
    {
      return Form.PAIRWISE_COMPARISON;
    }

    [...]
}
```

Listing 6 presents the tutorial's source code. It showcases how to create parameterized pairwise comparisons and display their string representations. Figure 5 illustrates the expected console output.

Listing 6: Tutorial's source code

```
// Store pieces of preferences in the list:
LinkedList<IPreferenceInformation>
    preferenceInformation = new LinkedList
    <>();

// Create the pairwise comparison result --
     preference:
preferenceInformation.add(
    PairwiseComparison.getPreference(new
    Alternative("A1", new double[]{1.0d, 0.0
    d}), new Alternative("A2", new double
    []{0.0d, 1.0d})));
// Create the pairwise comparison result --
     indifference:
preferenceInformation.add(
    PairwiseComparison.getIndifference(new
    Alternative("A3", new double[]{0.6d, 0.4
    d}), new Alternative("A4", new double
    []{0.4d, 0.6d})));
// Create the pairwise comparison result --
     incomparable:
preferenceInformation.add(
    PairwiseComparison.getIncomparable(new
    Alternative("A1", new double[]{0.7d, 0.5
    d}), new Alternative("A1", new double
    []{0.5d, 0.7d})));

// Display the preference information:
for (IPreferenceInformation pi :
    preferenceInformation)
{
  System.out.println(pi.toString());
  System.out.println("Form = " + pi.getForm
      () + "; is indirect = " + pi.
      isIndirect());
}
```

```
Pairwise comparison: A1 PREFERENCE A2
Form = PAIRWISE_COMPARISON; is indirect = true
Pairwise comparison: A3 INDIFFERENCE A4
Form = PAIRWISE_COMPARISON; is indirect = true
Pairwise comparison: A1 INCOMPARABILITY A1
Form = PAIRWISE_COMPARISON; is indirect = true

Process finished with exit code 0
```

Figure 5: Expected console output

### 2.7. *History of preference elicitation [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t1_concepts.t3_history***

During the recommendation-reaching process, various preference examples can be obtained from the decision maker. This framework introduces a dedicated class for collecting these pieces of information: *history.History*. One history object is instantiated for each *DecisionMakerSystem* (each *ModelSystem* also has its copy, which will be discussed later). Its central field is "*_preferenceInformation*", a list of all collected preference statements. As the JavaDoc states, this list keeps the preference examples from the oldest to the newest (the list will be validated each time it is modified). The list, however, does not collect the *IPreferenceInformation* objects directly. Instead, this interface's newly added object instance is wrapped using the *PreferenceInformationWrapper* class. This class is used when registering new preference information to supply additional metadata, e.g., unique ID (controlled by *History*), iteration number when collected, or the timestamp (also when collected). These additional data help ensure the correctness of all modifications of the "*_preferenceInformation*" list and provide valuable means for generating informative logs.

Listing 7 brings the tutorial's source code for convenience. It showcases how to create *History* object and iteratively supply it with three pre-defined pairwise comparisons. Finally, they are – iteratively – removed from history. After each step, the current state of the history of preference elicitation is printed. Note that operating on the *History* is typically the system's responsibility.

This tutorial just exhibits its basic functionalities. The expected console output is illustrated in Figure 6.

Listing 7: Tutorial's source code

```
 // Create preference information (dummy
     alternatives, i.e., not evaluated):
ArrayList<IPreferenceInformation>
    preferenceInformation = new ArrayList
    <>(3);
preferenceInformation.add(
    PairwiseComparison.getPreference(new
    Alternative("A1", 1), new Alternative("
    A2", 1)));
preferenceInformation.add(
    PairwiseComparison.getIndifference(new
    Alternative("A3", 1), new Alternative("
    A4", 1)));
preferenceInformation.add(
    PairwiseComparison.getIncomparable(new
    Alternative("A5", 1), new Alternative("
    A6", 1)));

// Create the history object:
History history = new History("History (
    tutorial)");
// Create criteria:

try
{
  // Go through iterations 0, 1,...,5.
  // Let's add the preference information
      in iteration = 0, 1, 2.
  // We will remove them in the remaining
      steps.
  for (int i = 0; i < 6; i++)
  {
    System.out.println("Iteration = " + i);

    // Register the preference information
        (creates the wrappers; see the
        return type):
    if (i < 3) history.
        registerPreferenceInformation(
        preferenceInformation.get(i), i,
        true);
    // ... or remove it (the list is small,
         so getting elements by the index
        does not hurt).
    else
    {
      // Get copy (note that the deep-copy
          is not guaranteed at the lower
          levels (e.g., implementations
      // of the IPreferenceInformation
          interface)
      LinkedList<
          PreferenceInformationWrapper>
          wrappers = history.
          getPreferenceInformationCopy();
      history.remove(wrappers.getFirst());
    }
    System.out.println(history.
        getFullStringRepresentation());
    System.out.println();
  }

} catch (HistoryException e)
{
  throw new RuntimeException(e);
}
```

### 2.8. *Preference model [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t1_concepts.t4_preference_model***

As with the preference information, generalizing various preference models proposed by the multi-criteria decision-aiding community is challenging from the programming perspective, as different models may impose a different processing flow. For instance, the so-called value models quantify the relevance of an alternative in an absolute sense using a scoring function. In turn, the relational models judge alternatives' relevance regarding how good they are when compared to another – hence, the evaluation is relative. Consequently, as with the preference information concept, the initial release of JECDM introduces very general interfaces and abstract classes to represent a preference model. Further, the primary focus of this framework is the value models, and the initial release includes one representative of this stream: $L_\alpha^w$-norm, which will be overviewed later.

Listing 8 illustrates some representative code snippets of the top-level *IPreferenceModel* interface (see the *model* package) that serves as a definition – or better to say, a header or signature – of a preference model (does not concern specific parameterization). It is a generic interface that accepts any generic type that implements *IInternalModel*. The latter is the starting point for designing so-called internal models, i.e., concrete models linked to their signatures. When it comes to model definition implementations, they all extend the *AbstractPreferenceModel* class that aggregates common fields and functionalities (model package). The instantiable extensions are available in *model.definitions* package. An example code snippet of the abstract class is brought in Listing 9. Noticeably, three common and self-explanatory fields are introduced: "*_name*" (model's name), "*_models*" (the internal models), "*_evaluator*" (auxiliary object to which the evaluation of alternatives is delegated; will be discussed later).

Listing 8: IPreferenceModel interface

```
/**
 * The top-level interface for preference
     models (objects responsible for
     mathematically modeling the decision
     maker's
 * preferences). The interface is generic
     and has to be parameterized with a type
      representing the internal model
```

```
Iteration = 0
History (tutorial)
[Pairwise comparison: A1 PREFERENCE A2 ; id = 0 ; iteration = 0 ; date time = [8.11.2024 18:12:44:25]]

Iteration = 1
History (tutorial)
[Pairwise comparison: A1 PREFERENCE A2 ; id = 0 ; iteration = 0 ; date time = [8.11.2024 18:12:44:25]]
[Pairwise comparison: A3 INDIFFERENCE A4 ; id = 1 ; iteration = 1 ; date time = [8.11.2024 18:12:44:46]]

Iteration = 2
History (tutorial)
[Pairwise comparison: A1 PREFERENCE A2 ; id = 0 ; iteration = 0 ; date time = [8.11.2024 18:12:44:25]]
[Pairwise comparison: A3 INDIFFERENCE A4 ; id = 1 ; iteration = 1 ; date time = [8.11.2024 18:12:44:46]]
[Pairwise comparison: A5 INCOMPARABILITY A6 ; id = 2 ; iteration = 2 ; date time = [8.11.2024 18:12:44:46]]

Iteration = 3
History (tutorial)
[Pairwise comparison: A3 INDIFFERENCE A4 ; id = 1 ; iteration = 1 ; date time = [8.11.2024 18:12:44:46]]
[Pairwise comparison: A5 INCOMPARABILITY A6 ; id = 2 ; iteration = 2 ; date time = [8.11.2024 18:12:44:46]]

Iteration = 4
History (tutorial)
[Pairwise comparison: A5 INCOMPARABILITY A6 ; id = 2 ; iteration = 2 ; date time = [8.11.2024 18:12:44:46]]

Iteration = 5
History (tutorial)
```

Figure 6: Expected console output

```
 * (e.g., L-norm, value function, etc.)
     that has to extend {@link
     AbstractInternalModel}.
 *
 * @author MTomczyk
 */
public interface IPreferenceModel<T extends
    IInternalModel>
{
  /**
   * Auxiliary method that can be used to
       register the current decision-making
       context {@link DMContext}.
   *
   * @param dmContext decision-making
       context
   * @throws PreferenceModelException the
       exception can be thrown and
       propagated higher
   */
  void registerDecisionMakingContext(
      DMContext dmContext) throws
      PreferenceModelException;

  /**
   * The main method for evaluating an
       alternative. Note that it is
       preferred to use {@link
       IPreferenceModel#evaluateAlternatives
       (AbstractAlternatives)}
   * as it may speed up calculations, may
       be the only implemented method, or
       can provide additional evaluation
       results
   * (implementation dependent).
   *
   * @param alternative the alternative to
       be evaluated
   * @return attained score
   * @throws PreferenceModelException the
       exception can be thrown and
       propagated higher
   */
  double evaluate(Alternative alternative)
      throws PreferenceModelException;

  /**
   * The main method for evaluating
       alternatives.
   *
   * @param alternatives alternatives to be
       evaluated
   * @return evaluation result object
   * @throws PreferenceModelException the
       exception can be thrown and
       propagated higher
   */
  EvaluationResult evaluateAlternatives(
      AbstractAlternatives<?> alternatives)
      throws PreferenceModelException;

  /**
   * Auxiliary method that allows
       determining the model preference
       direction (dedicated to value models;
       default
```

```
    * implementation = the flag is retrieved
        from the first internal model).
    *
    * @return true if smaller values are
       preferred, false otherwise
    */
  boolean isLessPreferred();

  [...]
}
```

Listing 9: AbstractPreferenceModel class

```
/**
 * Default, abstract implementation of {
    @link IPreferenceModel}. Contains
    common fields and functionalities.
 *
 * @author MTomczyk
 */
public abstract class
    AbstractPreferenceModel<T extends
    IInternalModel> implements
    IPreferenceModel<T>
{
  /**
   * The name of the preference model.
   */
  protected final String _name;

  /**
   * This array stores the internal models
      (extending {@link
      AbstractInternalModel}..
   */
  protected ArrayList<T> _models;

  /**
   * Auxiliary model for evaluating an
      alternative given the internal model
      (models) stored.
   */
  protected final IEvaluator<T> _evaluator;

  [...]
}
```

Let us now focus on the primary model implemented for the initial release of JECDM: weighted $L_\alpha^{\boldsymbol{w}}$-norm. It is a scalarizing function that is defined as follows:

$$L_\alpha^{\boldsymbol{w}}\left(\boldsymbol{s}^N\right) = \left[\sum_{i=1}^{M}\left(s_i^N w_i\right)^\alpha\right]^{\frac{1}{\alpha}}, \tag{1}$$

where $\alpha$ is a so-called compensation level ($\alpha \to$ the model is a weighted sum function; $\alpha = \infty \to$ the model is a weighted Chebyshev function), $M$ is the number of objectives, $\boldsymbol{w}$ is a weight vector of length $M$, and $\boldsymbol{s}^N$ is a solution (objective/performance) vector of length $M$ in the normalized space (follows the imposed standarization, i.e., $\mathbf{0}$-vector is a utopia point, while $\mathbf{1}$ is the nadir.

Introducing a model imposes providing its definition (implementation of the *IPreferenceModel* interface) and its internal representation (implementation of the *IInternalModel* interface; represents an actual formula). When it comes to the first, the framework introduces an *LNorm* class (*model.definitions* package), which is a straightforward implementation that fixes the generic type to a related internal model (*model.internals.value.LNorm*) and provides various customization means (see the code snippet in Listing 10). Note that the class extens a more general model *ScalarizingFunction* that can be coupled with any internal model that extends *AbstractScalarizingFunctionInternalModel*.

Listing 10: LNorm class (definition)

```
/**
 * Implementation of {@link
    IPreferenceModel} that uses L-norms to
    represent the decision maker's
    preferences.
 *
 * @author MTomczyk
 */
public class LNorm extends
    ScalarizingFunction<model.internals.
    value.scalarizing.LNorm> implements
    IPreferenceModel<model.internals.value.
    scalarizing.LNorm>
{
  [...]

  /**
   * Parameterized constructor.
   *
   * @param name          model's name
   * @param internalModel internal model
       used
   * @param evaluator     auxiliary model
       for evaluating an alternative given
       the internal model (models) stored
   */
  protected LNorm(String name, model.
      internals.value.scalarizing.LNorm
      internalModel, IEvaluator<model.
      internals.value.scalarizing.LNorm>
      evaluator)
  {
    super(name, internalModel, evaluator);
  }
}
```

When it comes to the *LNorms*'s internal model, it is *model.internals.value.LNorm* class. It follows a simple class hierarchy depicted in Figure 7.

The *LNorm* class extends *AbstractScalarFunctionInternalModel*, which wraps the *IScalarFunction* interface from the *space.scalarfunciton package* (*Utils* module). Overall, the top-level *IInternalModel* is highly consistent with such sources as *IScalarizingFunction*, meaning that ultimately it can serve as as a wrapper for some external implementations (such as im-

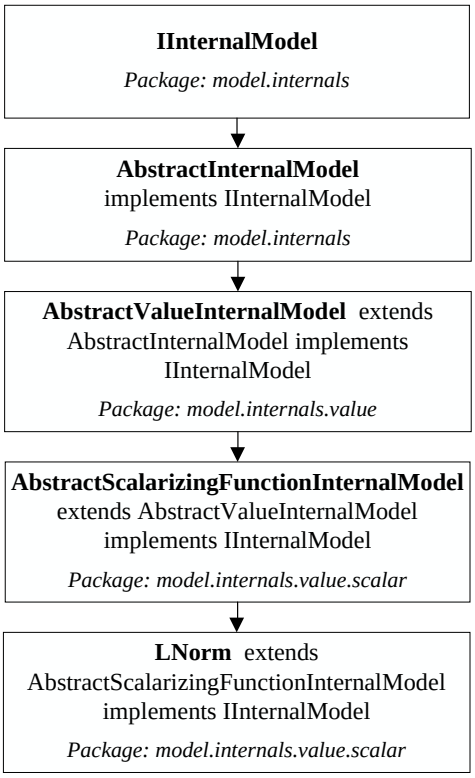



Figure 7: Class hierarchy (internal model)

plementations of the *IScalarizingFunction* interface). For instance, the interface allows reperameterizing the wrapped function – e.g., providing weight vectors, or supplying it with new normalization objects. Regarding the *LNorm* internal model, it wraps the *space.scalarfunction.LNorm* class from the *Utils* module. This class implements the function presented in Eq. 1. Naturally, the preference direction is "to minimize".

This tutorial's source codes (Tutorial4a and Tutorial 4b) showcase how to instantiate a preference model (*LNorm*) and employ it to evaluate a series of alternatives. Table 1 summarizes the data concerned by these sources for convenience. The first two columns present the two-element objective/performance vectors (there are 11 of them). Next, it is assumed that the known objective space bounds are explicitly spanned over these solutions. Hence, they are $[0, 4]$ and $[0, 2]$. It is also assumed that both criteria are of cost type. Therefore, the normalized objective vectors can be obtained simply by dividing the first performance score by 4 and the second by 2. The normalized objective vectors are provided in the following two columns. The next two columns are linked to two $L_\alpha^{\boldsymbol{w}}$-norms used in these tutorials. Both assume $\alpha = \infty$, hence these are Chebyshev functions. However, their internal weight vectors are different: $[0.7, 0.3]$ and $[0.5, 0.5]$. The numbers provided in these last two columns are scores attained by each normalized vector according to the respective $L_\alpha^{\boldsymbol{w}}$-norm (smaller values are preferred). Naturally, the first $L_\alpha^{\boldsymbol{w}}$-norm favors the first objective more than the second. According to it, the best solution is no. 3 (favors the first objective; represents the best balance, according to $[0.7, 0.3]$ weight vector). The second model incorporates equal weights. Consequently, according to this model, the normalized vector $[0.5, 0.5]$ attains the best score. The final columns is the average score attained by the normalized vector according to both models.

| No. | $s_1$ | $s_2$ | $s_1^N$ | $s_2^N$ | $L_\infty^{[0.7,0.3]}\left(\boldsymbol{s}^N\right)$ | $L_\infty^{[0.5,0.5]}\left(\boldsymbol{s}^N\right)$ | *Mean* |
|---|---|---|---|---|---|---|---|
| 1 | 0.0 | 2.0 | 0.0 | 1.0 | 0.30 | 0.50 | 0.40 |
| 2 | 0.4 | 1.8 | 0.1 | 0.9 | 0.27 | 0.45 | 0.36 |
| 3 | 0.8 | 1.6 | 0.2 | 0.8 | 0.24 | 0.40 | 0.32 |
| 4 | 1.2 | 1.4 | 0.3 | 0.7 | 0.21 | 0.35 | 0.28 |
| 5 | 1.6 | 1.2 | 0.4 | 0.6 | 0.28 | 0.30 | 0.29 |
| 6 | 2.0 | 1.0 | 0.5 | 0.5 | 0.35 | 0.25 | 0.30 |
| 7 | 2.4 | 0.8 | 0.6 | 0.4 | 0.42 | 0.30 | 0.36 |
| 8 | 2.8 | 0.6 | 0.7 | 0.3 | 0.49 | 0.35 | 0.42 |
| 9 | 3.2 | 0.4 | 0.8 | 0.2 | 0.56 | 0.40 | 0.48 |
| 10 | 3.6 | 0.2 | 0.9 | 0.1 | 0.63 | 0.45 | 0.54 |
| 11 | 4.0 | 0.0 | 1.0 | 0.0 | 0.70 | 0.50 | 0.60 |

Table 1: Data used in the tutorial's source code

**Tutorial4a** source code is brought in Listing 11 for convenience. It is a straightforward code showcasing how to define and supply the model with a single internal model ($L_\alpha^{\boldsymbol{w}}$-norm with weights = $[0.7, 0.3]$). Note that the model is supposed to employ an auxiliary implementation of *IEvaluator* interface. In short, its implementations should define the (alternatives, internal models) $\rightarrow$ *score*($s$) evaluation result (the result is stored in the supportive *EvaluationResult* object). Overall, the model definition delegates the evaluation of input alternatives to *IEvaluator*. This tutorial employs a simple *RepresentativeModel* evaluator, which assumes that there is just one internal model stored called "representative". This implementation delegates the evaluation of alternatives to the first internal model stored in *IPreferenceModel*. Consequently, this source code is expected to produce the scores reported in the third to last column in Table 1. Figure 8 illustrates the expected console output, which matches the expectations.

Listing 11: Tutorial's source code

```
// Create the model evaluator:
    representative = alternative will be
    evaluated as imposed by the first
    internal
// model stored (remaining ones will be
    ignored)
IEvaluator<LNorm> representative = new
    RepresentativeModel<>();

// Create the model definition:
IPreferenceModel<LNorm> model = new model.
    definitions.LNorm(representative);

// Create the internal model (
    parameterization):
LNorm lnorm = new LNorm(new double[]{0.7d,
    0.3d}, Double.POSITIVE_INFINITY,
new INormalization[]{new Linear(0.0d, 4.0d)
    , new Linear(0.0d, 2.0d)});

// Set the internal model (single model):
```

```
model.setInternalModel(lnorm);

// Create the artificial data (in the
    normalized space first, but the code
    then does the ''upscaling'' (so that
// the used normalization objects will be
    relevant).
int no = 11;
double[][] e = new double[11][2];
for (int i = 0; i < no; i++)
{
    e[i][0] = (double) i / (no - 1);
    e[i][1] = 1.0d - e[i][0];
    e[i][0] *= 4.0d; // upscale to original
        space
    e[i][1] *= 2.0d; // upscale to original
        space
}
ArrayList<Alternative> alternatives =
    Alternative.getAlternativeArray("A", e);

// Evaluate the alternatives and print the
    result:
try
{
    System.out.println("Model = " + model.
        getName());
    System.out.println("Less is better = "
        + model.isLessPreferred());
    EvaluationResult result = model.
        evaluateAlternatives(new
        Alternatives(alternatives));
    System.out.println(result.toString());

} catch (PreferenceModelException ex)
{
    throw new RuntimeException(ex);
}
```

```
Model = Preference model (L-norms)
Less is better = true
Evaluation time = 0 ms
Evaluation vector = 0,30 0,27 0,24 0,21 0,28 0,35 0,42 0,49 0,56 0,63 0,70
```

Figure 8: Expected console output

**Tutorial4b** extends the former code by employing the second *L-norm* and defining a custom evaluator (inner class) that calculates the alternative's final score as the average performance attained according to both internal models. The expected console output is illustrated in 9.

```
Model = Preference model (L-norms)
Less is better = true
Evaluation time = 0 ms
Evaluation vector = 0,40 0,36 0,32 0,28 0,29 0,30 0,36 0,42 0,48 0,54 0,60
```

Figure 9: Expected console output

## 3. Executing preference elicitation

This section discusses the *PreferenceElicitationModule* class – its processing flow and how to parameterize it and use it in practice. Note that the module is not to be instantiated manually. Instead, it is created by the top-level *DecisionSupportSystem* class upon the initialization. Nonetheless, this class will be discussed later because introducing it now would obscure readability. Hence, when discussing the module's functionalities, the reader is expected to assume the decision support system is already instantiated and some of its data is already accessible.

Before proceeding to code snippets, focus now on Figure 10. It illustrates a simplified processing flow imposed by the preference elicitation module. The following sections will delve into its steps more extensively, but let us briefly discuss them now to give the reader an overall understanding of this process.

First, in its current version, the decision support system does not have explicit control over when the preference elicitation process should be called. It is a prerogative of an external caller, e.g., an evolutionary algorithm. When the process is run, it starts with registering a so-called decision-making context (an external caller instantiates the context). This involves passing the object to all interested receivers. The context is a class that bridges the decision making system and the external caller, e.g., the evolutionary method. The classic decision-reaching process (i.e., not involving the optimization) operates in a quite sterile environment where all the alternatives are known in advance (usually). In turn, when implemented within an optimization process, the environment may constantly change. The context is intended to capture these changes, allowing the decision making system to understand what is happening. In the initial release, however, the context is a simple data container that passes data between the evolutionary method and the system. The most fundamental piece of data maintained is the alternative set, which – from this perspective – may be the optimizer's current population.

Next, the process checks whether the interaction with the decision maker(s) should be performed. This is controlled by a set of rules that the programmer can provide. If none of the rules favor the interaction, the process terminates. This involves unregistering (nulling) the context and returning a report on the process execution. In this case, the report will indicate that the interaction was not triggered; thus, the method has immediately returned.

If the interactions are to be performed, the process notifies the interested components that the "preference elicitation begins." Typically, the system handles such notifications. Nonetheless, the programmer will occasionally be required to deal with such a notification in their implementations. For instance, the already mentioned rules that decide whether to interact will be sent such a notification. Depending on the implementation, this information may be relevant for correct processing (e.g., the rule may favor the interaction, but how it may know whether it was indeed performed?).

In the case of the non-optimization environment, the processing could proceed directly to the actual interaction part. However, this module introduces a few pre-processing steps that

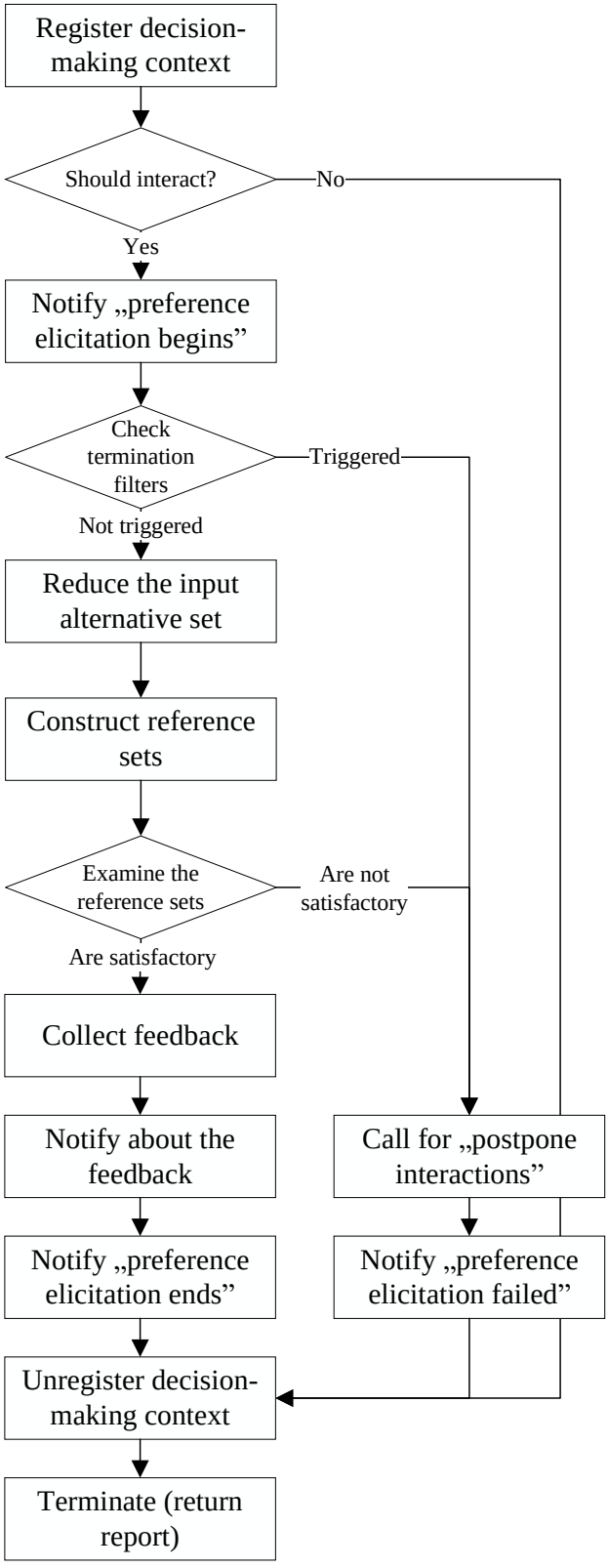



Figure 10: General scheme of the flow imposed by the preference elicitation module.

may (or may not) be used (may also be customized). They intend to alleviate the potential issues caused by the unpredictability of the shape of input data. For instance, the input alternatives (provided by the context) may be concentrated in an area that is too small to perform reliable interaction. If so, terminating the interaction process and moving back to the optimization would be more reasonable. A so-called termination filter(s) can trigger such a premature termination (multiple filters can be specified; one filter is enough for triggering the termination). The goal is to judge whether to proceed given the current context. If the filter opts for termination, the processing prepares for exit. For this reason, the interested receivers are provided a signal that the interactions should be postponed. This feedback is used, e.g., by the interaction rules to suitably adjust their processing (e.g., an implementation may then request trying to interact in the next call). After that, the notification about “preference elicitation failed” is sent to interested receivers. Then, the processing moves back to the “regular” termination path. It consists of notifying about the end of the preference elicitation, unregistering the decision-making context, and returning the report.

If the processing is not prematurely terminated, it may start performing the so-called reduction. The idea of this step is to filter out some unwanted input alternatives. The step can also be customized to a large extent by the programmer (e.g., the composition and the order of filters). For instance, the default implementation first removes duplicated alternatives (comparison is done in the objective/performance space). Then, it filters out dominated ones. This way, it is ensured that only unique and non-dominated alternatives are preserved before proceeding to the next steps.

The filtered, also called refined, alternatives are the major input for the next step – constructing reference sets. A reference set is a set of alternatives that is supposed to be evaluated/judged/presented to decision maker (or decision makers). The programmer is given here a great flexibility in adjusting this step. First, (s)he can specify one or more reference set constructors. Each constructor can be associated with all decision makers or just an explicitly appointed one. Next, it is supposed that the size of the returned set is adjustable (is a parameter) for each implemented constructor. Finally, one constructor is allowed to return multiple reference sets.

The result of the above process is then examined. First, it may turn out unsatisfactory. In the current implementation, it involves two scenarios:

- There were not enough alternatives in the refined set to construct at least one reference set (e.g., the refined set could consist of just one solution, while a pair of them had to be selected).
- No reference sets were constructed at all (e.g., due to improper parameterization.

If one of the above is the case, the process calls for “postpone interactions”, notifies about “preference elicitation failed”, unregisters decision-making context, and terminates.

If the reference sets have been constructed satisfactorily, the process begins collecting the feedback. This step intends to produce one feedback for one constructed reference set. Note that

the reference set may be appointed to some specific decision maker. Additionally, a real decision maker may associated with one so-called feedback provider object. Thus, the collected feedback objects may be stored in the decision makers' individual histories of preference elicitation (configuration dependent; it will be discussed more extensively when presenting source codes). Note that the current implementation assumes that the feedback will be produced given the constructed reference sets, i.e., there is no possibility for resignation (e.g., when the decision maker cannot judge the presented alternatives). Such a functionality will be implemented in one of the future updates.

If the feedback has been successfully collected, a notification about the new pieces of preference information will be sent. The implemented mechanisms will first store them in the decision makers' histories of preference elicitation (the process will correctly match the source of the feedback with the system). Then, the feedback will also be passed to decision makers' model systems. These systems additionally store individual copies of the preference elicitation history. The rationale is to allow the objects responsible for instantiating the preference model to manipulate this history without causing system errors. Such manipulations may be triggered, e.g., when trying to reinstate the model's consistency with the collected pieces of preferences. Following this step, the process notifies that "preference elicitation ends", unregisters the decision-making context, and terminates providing a complete report on the process execution.

### *3.1. Triggering preference elicitation [T]*

Source package: *t1_t10.t4_decision_support_module.t2_preference_elicitation_module.t1_interaction_trigger*

This tutorial section concerns the first step of the implemented preference elicitation process – checking whether the interaction with the decision maker(s) should be triggered. There are two central entities implemented for this purpose: the *IRule* interface that represents a rule for deciding whether to interact (see the *interaction.trigger.rules* package) and the *InteractionTrigger* class that maintain all the rules (see the *interaction.trigger* package).

#### *3.1.1. Rule for deciding when to interact*

Regarding the *IRule* interface, its most relevant code snippets are brought in Listing 12. Its primary method is "*shouldInteract*", which returns a yes/no (true/false) flag given the input decision-making context. Notably, the interface imposes the implementation of various callback/notification-like methods, e.g., "*postpone*" that is used to inform the rules that the triggered interaction has been postponed due to reasons provided via the *Postpone* object (*interaction.trigger.rules* package).

Listing 12: IRule interface

```
package interaction.trigger.rules;

import exeption.TriggerException;
import interaction.trigger.Postpone;
import dmcontext.DMContext;

/**
 * Interface for classes that are rules for
     deciding whether the interaction
    should be triggered.
 *
 * @author MTomczyk
 */
public interface IRule
{
  /**
   * Signature for the main method for
      deciding whether to interact.
   *
   * @param dmContext current decision-
      making context
   * @return true, if the rule favors
      interaction, false otherwise
   * @throws TriggerException the exception
       can be thrown and propagated higher
   */
  boolean shouldInteract(DMContext
      dmContext) throws TriggerException;

  /**
   * A call-back method that can be used to
       inform the object that the
      interaction, although triggered, is
      postponed
   * (e.g., due to the unavailability of
      alternatives).
   *
   * @param postpone object providing
      auxiliary data related to postponing
      (e.g., reason)
   */
  void postpone(Postpone postpone);

  [...]
}
```

When it comes to parameterizing the top-level *DecisionSupportSystem* object, the user will be requested to provide rule(s) for deciding whether to interact via the *InteractionTrigger* object. Listing 13 illustrates its most relevant code snippets. First, observe that the rules can be explicitly passed via the constructor. Second, its central method is "*shouldTriggerTheInteractions*." This procedure asks each rule whether to interact. The outcomes of this analysis are returned via a dedicated *Result* object (extends the already introduced *AbstractReport* class). The result contains a flag "*_shouldInteract*" that is self-explanatory. Moreover, it stores all the rules (their string representations) that opted for the interaction, which may be helpful when generating system logs.

Listing 13: InteractionTrigger class

```
/**
 * Main class for deciding whether to
    interact with the decision maker. The
    decision upon the interaction is made
```

```
     by
 * using a series of rules {@link IRule}.
     The interaction will be triggered if at
      least one rule favors interaction.
 * Note that all the rules objects are
     inspected when determining to interact
     (even if it is already known that the
 * interaction should be performed).
 *
 * @author MTomczyk
 */
public class InteractionTrigger
{
  /**
   * Series of rules for deciding whether
       to interact.
   */
  private final LinkedList<IRule> _rules;

  /**
   * Parameterized constructor.
   *
   * @param rules uses a series of rules
   */
  public InteractionTrigger(LinkedList<
      IRule> rules)
  {
    _rules = rules;
  }

  [...]

  /**
   * Main method for deciding whether to
       interact with the decision maker. The
        decision upon the interaction is
       made
   * by using a series of rules {@link
       IRule}. The interaction will be
       triggered if at least one rule favors
        interaction.
   * Note that all the rules objects are
       inspected when determining to
       interact (even if it is already known
        that the
   * interaction should be performed).
   *
   * @param dmContext current decision-
       making context
   * @return the decision wrapped in a
       result class ({@link interaction.
       refine.Result})
   * @throws TriggerException the exception
        can be thrown and propagated higher
   */
  public Result
      shouldTriggerTheInteractions(DMContext
       dmContext) throws TriggerException
  {
    Result r = new Result(dmContext);
    long pT = System.nanoTime();
    for (IRule rule : _rules)
    {
      if (rule.shouldInteract(dmContext))
      {
        r._shouldInteract = true;
        if (r._callers == null) r._callers
            = new LinkedList<>();
        r._callers.add(rule.
            getDetailedStringRepresentation
            ());
      }
    }
    r._processingTime = (System.nanoTime()
        - pT) / 1000000;
    return r;
  }

  [...]
}
```

The initial release of JECDM provides three default implementations of the *IRule* interface: *IterationInterval*, *TimeInterval*, and *Flag* classes, each extending a general *AbstractRule* class (see the *interaction.trigger.rule* package). Regarding the abstract class, it is a recommended starting point when creating own implementations (aggregates common functionalities, etc.). Nonetheless, these three classes are relatively versatile and thus usable in various implementations. Specifically, the *IterationInterval* defines interval-based patterns that trigger the interaction depending on the current iteration number (e.g., "opt for interactions every 10 iterations). Similarly, the *TimeInterval* class decides upon the interaction based on the time passed since the last interaction that ended successfully. Lastly, the *Flag* implementation maintains a volatile boolean flag that some external source can set. The flag is explicitly returned as the rule's decision. This class may be convenient when implementing some window-based application that is supposed to communicate with a real user who may wish to request the interaction. The following paragraphs showcase how to use these classes in practice.

***IterationInterval rule (Tutorial1a source code).*** The *IterationInterval* class is an iteration-based rule that establishes the pattern for interactions, which can be adjusted threefold:

- The starting iteration number can be provided (the interactions cannot start until the specified iteration number is reached).
- After the starting iteration number is reached, the interactions are requested at specified intervals (beginning from the starting iteration).
- The total number of iterations to be requested can be limited. Note that only the actually conducted interactions, i.e., those that ended successfully count.

Moreover, postponing the interactions will shift the request to the next iteration (even if it does not follow the specified pattern). However, suppose the following iteration already favors

interaction (as the established pattern implies). In that case, the missing request will be postponed even further (until there is some "available slot"). Additionally, the postponed request can also stack (e.g., when, for many iterations, there were no satisfactory conditions for interactions, but the rule already made many requests).

The tutorial's source code is brought in Listing 14 for convenience. It assumes the following scenario:

- The rule establishes the following pattern: the interactions should start from iteration no. 2, should be performed every 2 iterations, and they are limited to 3 iterations in total.
- There will be 10 iterations whose numbers follow the $0, 1, \ldots, 9$ pattern (current iteration numbers in the provided contexts).
- Consequently, the interactions should be requested in iterations no. 2, 4, and 6. However, it is assumed that postponing the interactions will be requested in iteration no. 2 and 4, effectively shifting them to the next iterations. Therefore, the rule will opt for interactions in iteration no. 3, 5 and 6.

As already explained, the rule is not to be executed manually in the established decision support system. Therefore, the provided source includes some additional, less relevant pieces of code that mimic using the rule within the preference elicitation module (calls for notification methods, etc.). Without them, the code would work incorrectly.

The code starts with creating the rule object (starting iteration = 2, interact every 2 iterations, limit for the number of successfully conducted interactions = 3). Then, then the loop simulates moving through subsequent iterations. Each iteration begins with creating a context. Note that is is almost empty – it holds only the current iteration number. Typically, the context must be completely filled, but this source code does not perform any validation (the *DecisionSupportSystem* class does). After creating the context, the rule is asked about whether to interact (note that most of the system's components can throw exceptions; hence a simple try-catch is involved, but the potential exception is not handled – it will not be thrown). If the answer is true, the code notifies the rule that the "preference elicitation begins". As defined by the scenario, the postponing will be triggered in iteration no. 2 and 4 (the *if* block). If this is the case, the *rule.postpone* method is called (consider examining the *Postpone* class). Then the rule is notified about the failure in the preference elicitation. Finally, the iteration ends by notifying the rule that the "preference elicitation ends". Figure 11 illustrates the expected console output – it is consistent with the expectations.

Listing 14: Tutorial1a: IterationInterval

```
// Create the rule (starting iteration = 2;
    interval = 2; limit = 2):
IRule rule = new IterationInterval(2, 2, 3)
    ;

// Work for 10 iterations:
for (int iteration = 0; iteration < 10;
    iteration++)
{
  // Typically, all the fields must be
      filled (but this simple tutorial does
      not check the validity):
  DMContext.Params pDMC = new DMContext.
      Params();
  pDMC._currentIteration = iteration;
  DMContext context = new DMContext(pDMC,
      null, null);

  try
  {
    boolean shouldInteract = rule.
        shouldInteract(context);
    System.out.println("Iteration = " +
        iteration + ": should interact = " +
         shouldInteract);

    if (shouldInteract)
    {
          // added to ensure correct
              processing
      rule.notifyElicitationBegins(context)
          ;

      // Simulate postponing interactions
          in iteration 2 and 4:
      if ((iteration == 2) || (iteration ==
          4))
      {
        System.out.println("Postponing
            interactions in iteration = " +
            iteration + " (current
            interaction is cancelled)");
        rule.postpone(new Postpone(Reason.
            COULD_NOT_DETERMINE_REFERENCE
            _SETS, iteration, LocalDateTime.
            now()));
        // added to ensure correct
            processing
        rule.
            notifyPreferenceElicitationFailed
            (context);
        }

        // added to ensure correct
            processing
        rule.notifyElicitationEnds(context)
            ;
    }

  } catch (TriggerException e)
      {
    throw new RuntimeException(e);
      }
}
```

***TimeInterval rule (Tutorial1b source code).*** The *TimeInterval* class establishes similar decision patterns as the *IteractionInter-*

```
Iteration = 0: should interact = false
Iteration = 1: should interact = false
Iteration = 2: should interact = true
Postponing interactions in iteration = 2
 (current interaction is cancelled)
Iteration = 3: should interact = true
Iteration = 4: should interact = true
Postponing interactions in iteration = 4
 (current interaction is cancelled)
Iteration = 5: should interact = true
Iteration = 6: should interact = true
Iteration = 7: should interact = false
Iteration = 8: should interact = false
Iteration = 9: should interact = false
```

Figure 11: Expected console output

*val* class. The difference is that the former considers the time that passed since the decision support system started (if no interaction has been reported yet) or the time that passed since the last successfully ended interaction. As with the iteration-based rule, the time-based counterpart can be parameterized with an initial delay for triggering the interactions (no sooner than), minimal time intervals between subsequent interactions (note that the rule does not call for interactions itself; hence the time should be considered "minimal"), and the limit for the number of successfully conducted interactions.

The source code – brought in Listing 15 is a straightforward reformulation of the code presented for the previous case (*IterationInterval*). The scenario involves simulating running the system for 6 seconds. Within this interval, 3 interactions should be requested. The first should not occur sooner than after 500 ms. The remaining two requests should be called no sooner than after 1500 ms each. Note that the rule requires supplying the context by the system starting timestamp (automatically supplied by the *DecisionSupportSystem* class). The expected console output is illustrated in Figure 12 (obviously, one may obtain slightly different timestamps).

Listing 15: Tutorial1b: TimeInterval

```
// Create the rule (starts after 1 second,
    perform every 2 seconds, max 3
    interactions):
IRule rule = new TimeInterval(500, 1500, 3)
    ;

long startTime = System.currentTimeMillis()
    ;
int iteration = 0;

// The system start timestamp must be
    provided via the context:
LocalDateTime systemStartTimestamp =
    LocalDateTime.now();

// Work for 6 seconds:
while ((System.currentTimeMillis() -
    startTime) < 6000)
{
  // Create the context:
  DMContext.Params pDMC = new DMContext.
      Params();
  pDMC._currentIteration = iteration++;
  DMContext context = new DMContext(pDMC,
      null, systemStartTimestamp);

  try
  {
    // Should interact?
    boolean shouldInteract = rule.
        shouldInteract(context);

    if (shouldInteract)
    {
          // If true: print the delta time
              and call proper notifications:
          System.out.println("Interaction
              triggered in " + (System.
              currentTimeMillis() -
              startTime) + " ms");
          rule. notifyPreferenceElicitation
               Begins(context);
          rule.
              notifyPreferenceElicitationEnds
              (context);
    }

  } catch (TriggerException e)
  {
    throw new RuntimeException(e);
  }
}
```

```
Interaction triggered in 514 ms
Interaction triggered in 2020 ms
Interaction triggered in 3520 ms
```

Figure 12: Expected console output

***Flag rule (Tutorial1c source code).*** The following tutorial showcases using the Flag *rule*. Since this is a simple wrapper for a boolean flag, the code is straightforward and introduces nothing new. The scenario considers going through $0, 1, \ldots, 9$ iteration numbers. The flag is set to true when the iteration number $\%3 = 0$. Otherwise, the flag is set to 0. Listing 16 presents the tutorial's source code, while Figure 13 depicts the expected console output.

Listing 16: Tutorial1c: Flag

```
// Create the rule (set the flag to false):
Flag rule = new Flag(false);

// Work for 10 iterations:
for (int iteration = 0; iteration < 10;
    iteration++)
{
  // Create the context
```

```
    DMContext.Params pDMC = new DMContext.
        Params();
    pDMC._currentIteration = iteration;
    DMContext context = new DMContext(pDMC,
        null, null);

    // Simulate changing the flag (true for
        every 3 iterations):
    rule.setFlag(iteration % 3 == 0);

    try
    {
      boolean shouldInteract = rule.
          shouldInteract(context);
      if (shouldInteract)
      {
        rule.
            notifyPreferenceElicitationBegins(
            context);
        rule.notifyPreferenceElicitationEnds(
            context);
      }

      System.out.println("Iteration = " +
          iteration + ": should interact = " +
           shouldInteract);
    } catch (TriggerException e)
        {
      throw new RuntimeException(e);
    }
}
```

```
Iteration = 0: should interact = true
Iteration = 1: should interact = false
Iteration = 2: should interact = false
Iteration = 3: should interact = true
Iteration = 4: should interact = false
Iteration = 5: should interact = false
Iteration = 6: should interact = true
Iteration = 7: should interact = false
Iteration = 8: should interact = false
Iteration = 9: should interact = true
```

Figure 13: Expected console output

### 3.1.2. *Interaction trigger*

***Using multiple rules (Tutorial1d source code).*** When configuring the decision support system, not the rules but the *InteractionTrigger* object must be supplied. It maintains all the rules and ultimately decides whether to interact by questioning them. This tutorial shows how to create the trigger and use it in practice. Since the code is straightforward, it is not presented here for conciseness. Listing 17 presents only the code snippet focused on object creation. This scenario also considered going through $0, 1, \ldots, 9$ iterations. As the code implies, two *IterationInterval* rules simultaneously decide upon the interactions. The first opts for triggering the interactions every 4 iterations, starting from iteration number 1 and limiting the number of requests to 3. The second opts for interactions every 5 iteration. Consequently, the interactions should be triggered in iterations number 0 (favored by the second rule), 1 (favored by the first rule), and 5 (favored by both rules). Figure 14 illustrates the expected console output. This time, the string representations of the results returned by the *InteractionTrigger* are printed as convenient logs. These contain practical data, e.g., which rules were opted for during the interaction.

Listing 17: Tutorial1d: InteractionTrigger

```
// Create the rules:
LinkedList<IRule> rules = new LinkedList
    <>();
rules.add(new IterationInterval(1, 4, 2));
rules.add(new IterationInterval(5));

// Create the interaction trigger:
InteractionTrigger interactionTrigger = new
     InteractionTrigger(rules);
```

```
Current iteration = 0
    Should interact = true
    Processing time  = 4 ms
    Iteration = 0
    Date and time (decision-making context) = [6.11.2024 11:14:35:374]
    Date and time (report creation) = [6.11.2024 11:14:35:374]
    Reasons for interaction:
     - ITERATION_INTERVAL_0_5
Current iteration = 1
    Should interact = true
    Processing time  = 0 ms
    Iteration = 1
    Date and time (decision-making context) = [6.11.2024 11:14:35:402]
    Date and time (report creation) = [6.11.2024 11:14:35:402]
    Reasons for interaction:
     - ITERATION_INTERVAL_1_4
Current iteration = 2
Current iteration = 3
Current iteration = 4
Current iteration = 5
    Should interact = true
    Processing time  = 0 ms
    Iteration = 5
    Date and time (decision-making context) = [6.11.2024 11:14:35:402]
    Date and time (report creation) = [6.11.2024 11:14:35:402]
    Reasons for interaction:
     - ITERATION_INTERVAL_1_4
     - ITERATION_INTERVAL_0_5
Current iteration = 6
Current iteration = 7
Current iteration = 8
Current iteration = 9
```

Figure 14: Expected console output

## 3.2. *Analyzing and refining the alternatives [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t2_preference_elicitation_module.t2_refiner***

The termination check and the reduction steps (illustrated in Figure 10) are handled by the *Refiner* class. As the name indicates, its responsibility is to refine, i.e., preprocess the input alternative sets. For this reason, it maintains termination and reduction filters. The following sections will first discuss them before showcasing how to use the *Refiner* class. Note that the overall process imposed constitutes a pipeline that operates on an alternative array (*AbstractAlternatives*).

### *3.2.1. Termination filter*

The termination filter is a "safety switch", i.e., it decides whether to prematurely terminate the ongoing preference elicitation process due to unfavorable conditions. The *ITerminationFilter* interface is brought in Listing 18 (see *interaction.refine.filters.termination* package). It is a relatively straightforward interface that introduces a single method signature that is supposed to return a results-like object given the input context.

Listing 18: ITerminationFilter interface

```
/**
 * Interface for termination filters (
    decide whether to end the preference
    elicitation process prematurely).
 *
 * @author MTomczyk
 */
public interface ITerminationFilter
{
  /**
   * The method verifying whether the
      reference sets construction process
      should be terminated.
   *
   * @param dmContext          current
      decision-making context
   * @return filter's indications
   * @throws RefinerException exception can
      be thrown and propagated higher
   */
  TerminationResult shouldTerminate(
     DMContext dmContext) throws
     RefinerException;
}
```

***IRequiredSpread filter (Tutorial2a source code)***. The initial release of JECDM provides a single implementation of the *ITerminationFilter* interface: **RequiredSpread** class (it is recommended to use the *AbstractTerminationFilter* class when implementing own filters). It analyses the spread of the input alternatives superset (provided via the context). The termination will be triggered if all the alternatives occupy an area in the normalized objective/performance space that is too dense. The rationale is to avoid making unrealistic interactions, i.e., how to interact when all alternatives occupy practically a single point?). Another motivation is to avoid forcing the system to operate on complicated data – learning preferences may be complicated when the alternatives judged by the decision maker differ only insignificantly.

The class calculates the spans (min/max values) of alternatives' objective/performance vectors given each dimension (criterion/objective). The spans are calculated in normalized space – the normalization functions are derived from the context. If all the spans are smaller than the specified thresholds, the filter requests termination. In turn, exceeding the threshold given one dimension is enough to reject termination.

The tutorial's source code considers the following scenario:

- There are two iterations: 0 and 1.
- The objective space is spanned over $[0, 2]^2$ bounds, and the (two) criteria are of cost type. Consequently, the input objective vectors' values will be divided by two when creating normalized vectors.
- The termination filter assumes a common threshold of 0.1 for both objectives (in practice, a significantly smaller threshold is recommended; e.g., the default parameterization of *Refiner* recommends the threshold of 0.0001).
- The alternatives' performance vectors ($[f_1, f_2]$) for both iterations are presented in Table 2.

| $f_1$ | $f_2$ |
|---|---|
| Iteration = 0 | |
| 2.0 | 1.4 |
| 1.0 | 1.2 |
| 0.6 | 1.0 |
| 0.2 | 0.2 |
| 1.0 | 0.4 |
| Iteration = 1 | |
| 0.0 | 0.2 |
| 0.1 | 0.3 |
| 0.19... | 0.39... |

Table 2: Objective vectors used in the tutorial's source code

Given the above conditions, the termination is expected to be requested only in iteration 1 – the spans are $[0; 0.0999...]$ and $[0.1; 0.1999...]$, hence each is smaller than the threshold of 0.1 (note that the spans are calculated in the normalized space). The source code is straightforward; hence, it is not brought here for conciseness. However, Figure 15 presents the expected console output. Note that the termination result object also provides an informative message that can be used as a log. It tells that the termination is not requested in iteration 0 because the threshold was exceeded at criteria C0 and C1.

```
Should terminate = false; message = Threshold exceeded or equal at criteria = C0 C1
Should terminate = true; message = The set is invalid (not sufficiently spread)
```

Figure 15: Expected console output

### 3.2.2. *Reduction filter*

The reduction filters are responsible for excluding less promising alternatives from processing. They implement the *IReductionFilter* interface, showcased in Listing 19 (*interaction.refine.reduction* package). The "*reduce*" method accepts an array of alternatives to be inspected and returns a filtered array (preferably a new object). The general *Refiner* object can be supplied with multiple ordered reducers, forming a filtering pipeline (the output of one filter is the input for the next in order).

Listing 19: IReductionFilter interface

```
/**
 * Interface for objects responsible for
    reducing the superset.
 *
 * @author MTomczyk
 */
public interface IReductionFilter
{
  /**
   * The main method that filters out not
      valid alternatives from the super set
      .
   *
   * @param dmContext          current
      decision-making context
   * @param processedSet upper (super) set
   * @return alternatives that passed the
      filter tests (are valid and can be
      processed further), it should be a
      subset of the input set (recommended
      to make it a different object than
      the input)
   * @throws RefinerException exception can
       be thrown and propagated higher
   */
  AbstractAlternatives<?> reduce(DMContext
     dmContext, AbstractAlternatives<?>
     processedSet) throws RefinerException;
}
```

The initial release of JECDM provides two useful reduction filters: *RemoveDominated* implementation that filters out dominated alternatives, and *RemoveDuplicatesInOS* that removes duplicated alternatives (comparison is done in the objective/performance space, an alternative to be left is selected arbitrarily). It is recommend to use the *AbstractReductionFilter* when implementing your own filters.

***Reduction filters (Tutorial2b source code).*** This tutorial's source code showcases how to employ the *RemoveDuplicatesInOS* and *RemoveDominated* filters sequentially in the given order. The scenario considers two-dimensional space with criteria/objectives to be minimized. Table 3 brings the input alternatives array for convenience. The *RemoveDuplicatesInOS* preserves the first in-order duplicated alternative while removing the rest. Consequently, the expected output of this first filter is $[[0.0, 0.6], [0.1, 0.1], [0.5, 0.4], [0.3, 0.0], [0.3, 0.2], [0.1, 0.3]]$. Then, the second filter should return only three alternatives: $[[0.0, 0.6], [0.1, 0.1], [0.3, 0.0]]$. As for the confirmation, Figure 16 illustrates the expected console output of this source code.

| $f_1$ | $f_2$ | Duplicated? | Dominated? |
|---|---|---|---|
| 0.0 | 0.6 | Y | |
| 0.1 | 0.1 | Y | |
| 0.5 | 0.4 | | Y |
| 0.0 | 0.6 | Y | Y |
| 0.3 | 0.0 | | |
| 0.1 | 0.1 | Y | |
| 0.3 | 0.2 | | Y |
| 0.1 | 0.3 | | Y |

Table 3: Objective vectors used in the tutorial's source code

```
Remain (after removing duplicates):
A0: 0,00 0,60
A1: 0,10 0,10
A2: 0,50 0,40
A4: 0,30 0,00
A6: 0,30 0,20
A7: 0,10 0,30
Remain (after removing dominated):
A0: 0,00 0,60
A1: 0,10 0,10
A4: 0,30 0,00
```

Figure 16: Expected console output

### 3.2.3. *Refiner*

As already explained, the *Refiner* was designed to control the execution of the termination and reduction filters. Listing 20 provides its code snippet focused on the params container used to parameterize the object instance. Its two essential fields are, as expected, termination and reduction filters. They are stored in the list. Note that the list can be null, meaning the respective filters will not be used. Further, the order of stored reduction filters matters as they will be executed precisely in this provided order, and the output of one filter will serve as the input for the next one. Finally, the class introduces static means for getting the default parameterization of the params container (or the *Refiner* itself). The default object instance involves the *RequiredSpread* termination filter and two reduction filters: *RemoveDuplicatesInOS* and *RemoveDominated* (in this order). Such configuration is versatile and enhances further processing steps (e.g., it is more convenient when it is ensured that the maintained alternatives are unique and non-dominated). Note that the primary method of this class is "*refine*", and it returns a report-like complex object that summarizes the conducted refining process.

Listing 20: Refiner class (params container)

```
/**
 * The class can indicate premature
    termination or reduce the input set
    given selected conditions. Finally, a
    subset of
 * the input superset will serve as a basis
    for reference set construction (see {
    @link ReferenceSetsConstructor}).
 *
 * @author MTomczyk
 */
public class Refiner
{
  /**
   * Params container.
   */
  public static class Params
  {
    /**
     * Optional termination filters (can be
        null).
     */
    public LinkedList<ITerminationFilter>
        _terminationFilters;

    /**
     * Optional reduction filters (can be
        null = not used). Note that the
        order matters: the filters will be
     * executed in the given order, and the
        output of one filter will be the
        input for the next one.
     */
    public LinkedList<IReductionFilter>
        _reductionFilters;

    /**
     * Creates default object instance (
        default parameterization). It first
        instantiates the termination
        filters
     * using {@link RequiredSpread} (
        threshold of 0.001 is used). Next,
        reduction filters involve
     * {@link RemoveDuplicatesInOS} and {
        @link RemoveDominated} (in the
        provided order).
     *
     * @return default object instance
     */
    public static Params getDefault()
    {
      return getDefault(0.0001d);
    }


    /**
     * Creates default object instance (
        default parameterization). It first
        instantiates the termination
        filters
     * using {@link RequiredSpread}. Next,
        reduction filters involve {@link
        RemoveDuplicatesInOS} and
     * {@link RemoveDominated} (in the
        provided order).
     *
     * @param thTermination the threshold
        for the termination filter
     * @return default object instance
     */
    public static Params getDefault(double
        thTermination)
    {
      Params p = new Params();
      p._terminationFilters = new
          LinkedList<>();
      p._terminationFilters.add(new
          RequiredSpread(thTermination));
        p._reductionFilters = new
            LinkedList<>();
      p._reductionFilters.add(new
          RemoveDuplicatesInOS());
      p._reductionFilters.add(new
          RemoveDominated());
      return p;
    }
  }

  [...]
}
```

***Refiner (Tutorial2c source code).*** This tutorial's source code shows how to create and use the *Refiner* object. It uses the default parameterization, and the scenario setting is the same as in the previous tutorial on the reduction filters (hence, the input alternatives are the same as in Table 3). Figure 17 illustrates the expected console output, which is a string representation of the result of the refining process (termination was not requested; three alternatives remained after the reduction step).

```
Status = PROCESS_ENDED_SUCCESSFULLY
Iteration = 0
Date and time (decision-making context) = [6.11.2024 16:26:57:889]
Date and time (report creation) = [6.11.2024 16:26:57:891]
Terminated due to termination filter = false
Termination filters processing time = 2 ms
Reduction size = 5
Reduction filters processing time = 2 ms
Final alternatives = A0, A1, A4
```

Figure 17: Expected console output

### 3.3. Constructing reference sets [T]

Source package: *t1_t10.t4_decision_support_module.* ***t2_preference_elicitation_module.*** ***t3_reference_sets_constructor***

The refining process is followed by constructing so-called reference sets in the *PreferenceElicitationModule* implementation. A reference set is a composition of preselected alternatives (see the *ReferenceSet* class in the *interaction.reference* package). These are picked, by default, from the refined alternatives set. The reference set is supposed to be presented to the decision maker for critical evaluation. Overall, the system assumes that produced sets will be shown to appointed decision maker, and feedback object will be returned (the specific classes representing the feedback and the decision maker will be discussed later). There are two central sources in this context that are discussed in the following sections: *IReferenceSetConstructor* interface – expected to produce a reference set from refined alternatives, and *ReferenceSetsConstructor* – the main class that manages the related processing.

### *3.3.1. Reference set constructor interface*

The *IReferenceSetConstructor* is brought in Listing 21 for convenience. One can implement it to introduce one's own reference set constructor. For this reason, the "*getExpectedSize*" method must be implemented. It should inform the caller about the number of alternatives supplied via a single *ReferenceSet* returned by the primary method "*constructReferenceSets*". These sizes are inspected when moving from the refining phase to constructing reference sets. It may turn out that the number of refined alternatives is smaller than the minimal known expected size. If so, a premature termination will be triggered as there are not enough refined alternatives to construct all requested reference sets.

Regarding the "*constructReferenceSets*", it is supposed to return a list of constructed reference sets. By default, it is assumed that the reference sets will be formulated from the input filtered/refined alternatives (obviously, the array is considered read only).

The initial release of JECDM provides two default implementations of the *IReferenceSetConstructor* interface: *AllAlternatives* and *RandomPairs* (note that there is also a third implementation: *PWI*, but it will be discussed later). These classes can be found in *interaction.reference.constructor* package and will be examined in the following sections.

Listing 21: IReferenceSetConstructor interface

```
/**
 * Interface for classes responsible for
    preparing a reference set (or sets)
    given the input filtered alternatives.
 *
 * @author MTomczyk
 */
public interface IReferenceSetConstructor
{
  /**
   * Implemented method should notify about
       the reference set size the main
      construction procedure will create.
      The preference elicitation module
      will trigger the termination if there
       are not enough refined alternatives
      to produce the reference set of the
      specified size.
   *
   * @param filteredAlternatives filtered
      alternatives (passed the termination
      and reduction steps)
   * @return expected size of the reference
       set
   * @throws
      ReferenceSetsConstructorException the
       exception can be thrown and
      propagated higher
   */
  int getRequiredSize(AbstractAlternative
      <?> filteredAlternatives) throws
      ReferenceSetsConstructorException;

  /**
   * Main method for constructing a
      reference set (or sets). The method
      is supposed to return null if no
      valid
   * reference sets can be constructed (the
       elements of the output list cannot
      be null)
   *
   * @param dmContext
      current decision-making context
   * @param filteredAlternatives filtered
      alternatives (passed the termination
      and reduction steps)
   * @return constructed reference set (or
      sets); should return null if no such
      sets were generated (the elements
      cannot be null)
   * @throws
      ReferenceSetsConstructorException the
       exception can be thrown and
      propagated higher
   */
  LinkedList<ReferenceSet>
      constructReferenceSets(DMContext
      dmContext, AbstractAlternative<?>
      filteredAlternatives) throws
      ReferenceSetsConstructorException;

  [...]
}
```

***AllAlternatives (Tutorial3a source code).*** The *AllAlternatives* class was designed to explicitly return the input refined set as it is. The rationale for using such a constructor may be implementing some window-based application that could present the decision maker with all alternatives as a list. Then, it will be up to them to judge some of these, e.g., formulate pairwise comparisons. The implementation is relatively straightforward and is brought in Listing 22 for convenience. When it comes to "*getSize*" method, it returns the size of the input "*filteredAlternatives*". Regarding the "*constructReferenceSets*" method,

it wraps the input alternatives set in a *ReferenceSet* (note that the constructor copies the provided objects) and returns it as a single-element list.

For convenience, this tutorial's source code was brought in Listing 23. It is a straightforward code that showcases how to instantiate the *AllAlternatives* class and what results it produces. The expected console output is brought in Figure 18. As expected, the *ReferenceSet* object returned by *AllAlternatives* contains the same alternatives that were processed by the reference set constructor.

Listing 22: AllAlternatives class

```
/**
 * Implementation of {@link
    IReferenceSetConstructor} for
    constructing a single, general set that
     wraps all the filtered alternatives.
 *
 * @author MTomczyk
 */
public class AllAlternatives implements
    IReferenceSetConstructor
{
  /**
   * The method returns back the size of
       the input filtered alternatives set.
   *
   * @param filteredAlternatives filtered
       alternatives (passed the termination
       and reduction steps)
   * @return size of the input filtered
       alternatives set
   * @throws
       ReferenceSetsConstructorException the
        exception can be thrown and
       propagated higher
  */
  @Override
  public int getExpectedSize(
      AbstractAlternatives<?>
      filteredAlternatives) throws
      ReferenceSetsConstructorException
  {
    if (filteredAlternatives == null)
    throw new
        ReferenceSetsConstructorException("
        The filtered alternatives array is
        null", this.getClass());
    return filteredAlternatives.size();
  }

  /**
   * The method constructs a single,
       general set that wraps all the
       filtered alternatives.
   *
   * @param dmContext  current decision-
       making context
   * @param filteredAlternatives filtered
       alternatives (passed the termination
       and reduction steps)
   * @return wrapped input filtered
       alternatives
   * @throws
       ReferenceSetsConstructorException the
        exception can be thrown and
       propagated higher
   */
  @Override
  public LinkedList<ReferenceSet>
      constructReferenceSets(DMContext
      dmContext, AbstractAlternatives<?>
      filteredAlternatives) throws
      ReferenceSetsConstructorException
  {
    LinkedList<ReferenceSet> r = new
        LinkedList<>();
    r.add(new ReferenceSet(
        filteredAlternatives));
    return r;
  }

  [...]
}
```

Listing 23: Tutorial's source code

```
// All alternatives = treats the input
    refined alternatives as the reference
    set (useful for practical applications)
IReferenceSetConstructor allAlternatives =
    new AllAlternatives();

// Create the context:
DMContext.Params pDMC = new DMContext.
    Params();
pDMC._currentIteration = 0;
DMContext context = new DMContext(pDMC,
    null, null);

// Input refined alternatives:
AbstractAlternatives<?> refinedAlternatives
     = new Alternatives(Alternative.
    getAlternativeArray("A", new double[][]{
  {1.0d, 0.1d}, {0.2d, 2.0d}, {0.1d, 1.0d}
 }));

try
{
  LinkedList<ReferenceSet> referenceSets =
      allAlternatives.constructReferenceSets
      (context, refinedAlternatives);

  // Number of reference sets (should be
      one):
  System.out.println("No. reference sets =
      " + referenceSets.size());
  ReferenceSet rs = referenceSets.getFirst
      ();
  System.out.println(rs.toString());

} catch (ReferenceSetsConstructorException
    e)
```

```
{
  throw new RuntimeException(e);
}
```

```
No. reference sets = 1
Alternatives = A0, A1, A2
```

Figure 18: Expected console output

***RandomPair (Tutorial3b source code)***. The *RandomPairs* class, as the name indicates, is supposed to generate random pairs of alternatives from the input filtered set. It extends the common *AbstractPairsConstructor* class that provides common fields and functionalities (in the initial release of JECDM, it has only two extensions: *RandomPairs* and *PWI*). The extensions of *AbstractPairsConstructor* are useful when it comes to creating an interactive system that occasionally can ask the decision maker to compare a pair of solutions pairwise – and the preselection of the alternatives to be compared is done by the algorithm itself. The central parameters (fields) of *RandomPairs* implementation are:.

- "**int _pairs**": This parameter represents the number of pairs to be constructed from the input filtered set of alternatives.
- "**IValidator[] _validators**": The extensions *AbstractPairsConstructor* can analyze a constructed pair given some specified requirements. If a pair does not satisfy these conditions, it should be rejected, and the process of seeking a valid pair should continue. *IValidator* (*interaction.reference.validator* package) introduces a simple contract "*boolean isValid(DMContext dmContext, Alternative a1, Alternative a2)*". The method returns a flag true/false, indicating the validity of a pair being inspected. In the context of *RandomPairs*, the implementation randomly draws a pair of alternatives and checks the validity. If the test fails, the process is repeated (repeated as long as no valid pair is identified). Thus, the process implements a Monte Carlo sampling. Note that the reference set constructor can return null if it cannot construct valid sets.
- "**int _noHitsBeforeCycling**: *RandomPairs* generates two alternatives from the input filtered set randomly from a uniform distribution, and then it tests the validity of generated in this way pair. However, many pairs may be invalid due to employed validators, making the Monte Carlo sampling highly ineffective. This implementation switches to a "cyclic model" after a specified number of improper pairs are constructed to resolve this issue. This field specifies the threshold, and the default value is 10. "Cyclic mode" means that the method will generate the candidate pairs not randomly but by incrementally shifting the indices of selected alternatives (the pointer will move back to 0 after reaching the alternatives limit). The process ends successfully when a valid pair is found or terminates if no valid pair is identified after examining all possible pairs of alternatives.

This tutorial's source code presets *RandomPair* in practice. The scenarios are as follows:

- There are four alternatives $A0$ – $A3$: $[2.0, 0.2]$, $[0.5, 0.3]$, $[0.4, 0.4]$, $[0.2, 2.0]$.
- It is assumed that the known objective space is $[0, 2]^2$.
- *RandomPairs* is used to generate 20000 pairs (called 10000 times, each time producing two pairs), and the distribution matrix is constructed to inspect how many times different pairs were constructed.
- A pair is considered invalid if the coordinate differences for all objectives (in the normalized space) are smaller than 0.1. For this reason, a *RequiredSpread* validator with the threshold of 0.1 is used (this validator wraps the *RequiredSpread* termination filter). This validator can be useful for practical applications: it prevents selecting alternatives that are too similar for comparison.

Given the above specification, the pair $[A1; A2]$ is not valid (normalized $A1 = [0.25, 0.15]$; normalized $A2 = [0.2, 0.2]$; thus, the coordinate differences are 0.05 and 0.05], and are smaller than 0.1). Obviously, pairs consisting of the same alternatives also cannot be generated.

The tutorial's source code is relatively straightforward and, thus, not presented here. However, Figure 19 illustrates the expected console output. This printed matrix shows how many times different pairs of alternatives were picked (row = the first alternative in a pair; column = the second). As expected, the same alternatives, i.e., $AX$ ($X = 0, 1, 2, 3$), were not used to form any pair. Additionally, pairs $[A2, A3]$ and $[A3, A2]$ were also excluded. Finally, the remaining pairs were selected (practically) the same number of times – which was expected as the *RandomPairs* aims to create pairs from a uniform distribution.

```
0 1943 1942 2063
1984 0 0 2012
1983 0 0 2023
2015 1998 2037 0
```

Figure 19: Expected console output

### *3.3.2. Reference sets constructor class*

**Tutorial3c** finally presents how to instantiate the *ReferenceSetsConstructor* class. Its instance is intended to be supplied by the user when creating the decision support system. It generally maintains provided implementations of *IReferenceSetConstructor* and uses them to create final reference sets when requested. However, *ReferenceSetsConstructor* allows mapping the reference set constructors with all decision makers, or some specific ones, allowing this way to execute some decision maker-dependent processing.

The tutorial's source code assumes the following scenario:

- Two decision makers participate in the decision-reaching process: DM1 and DM2.
- Refiner produces four alternatives: *A*0-*A*3.
- *AllAlternatives* constructor is appointed to all decision makers.
- Additionally, each decision maker is appointed a dedicated *RandomPairs* constructor (set to construct one pair).

Listing 24 showcases the tutorial's code snippet dedicated to instantiating *ReferenceSetsConstructor* object for convenience. Finally, Figure 19 illustrates the expected console output – an informative log on the constructed sets. The log states that (i) one 4-element common set was constructed (appointed to all decision makers), and two 2-element reference sets (random pairs) were built – each dedicated to a different decision maker.

Listing 24: Tutorial's source code

```
 // Create the params container:
ReferenceSetsConstructor.Params pRSC = new
    ReferenceSetsConstructor.Params();
pRSC._commonConstructors = new LinkedList
    <>();
// Use GlobalSet for all DMs:
pRSC._commonConstructors.add(new
    AllAlternatives());

// Create DMs identifiers (strings):
String[] DMs = new String[]{"DM1", "DM2"};

// Create reference sets constructors for
    all DMs (RandomParis, each DM has a
    dedicated constructor):
pRSC._dmConstructors = new HashMap<>();
{
    LinkedList<IReferenceSetConstructor>
        constructors = new LinkedList<>();
    constructors.add(new RandomPairs(new
        RequiredSpread(0.1d)));
    pRSC._dmConstructors.put(DMs[0],
        constructors);
}
{
    LinkedList<IReferenceSetConstructor>
        constructors = new LinkedList<>();
    constructors.add(new RandomPairs(new
        RequiredSpread(0.1d)));
    pRSC._dmConstructors.put(DMs[1],
        constructors);
}

// Create the reference sets constructor:
ReferenceSetsConstructor RSC = new
    ReferenceSetsConstructor(pRSC);
```

```
Status = PROCESS_ENDED_SUCCESSFULLY
Iteration = 0
Date and time (decision-making context) = [11.11.2024 15:1:55:996]
Date and time (report creation) = [11.11.2024 15:1:56:0]
Terminated due to having not enough alternatives = false
Terminated due to having not enough alternatives message = null
Processing time = 5 ms
Results of the refining process:
  Status = PROCESS_ENDED_SUCCESSFULLY
  Iteration = 0
  Date and time (decision-making context) = [11.11.2024 15:1:55:996]
  Date and time (report creation) = [11.11.2024 15:1:55:999]
  Terminated due to termination filter = false
  Termination filters processing time = 0 ms
  Reduction size = 0
  Reduction filters processing time = 0 ms
  Final alternatives = A0, A1, A2, A3
Reference sets container:
  Common reference sets:
    Constructed reference sets = 1
    Total number of unique sizes = 1
    There is 1 reference set with size = 4
    Printing reference set no. 0
      Alternatives = A0, A1, A2, A3
  Reference sets unique to each decision maker:
  Reference sets unique to DM1:
    Constructed reference sets = 1
    Total number of unique sizes = 1
    There is 1 reference set with size = 2
    Printing reference set no. 0
      Alternatives = A0, A2
  Reference sets unique to DM2:
    Constructed reference sets = 1
    Total number of unique sizes = 1
    There is 1 reference set with size = 2
    Printing reference set no. 0
      Alternatives = A2, A3
```

Figure 20: Expected console output

### 3.4. Generating a feedback [T]

Source package: *t1_t10.t4_decision_support_module.* ***t2_preference_elicitation_module.*** ***t4_feedback_provider***

After the reference sets are constructed (and validated), the preference elicitation module collects the feedback. This step is handled by the "*FeedbackProvider*" class (*interaction.feedbackprovider* package). Listing 25 provides its params container (most essential code snippets). Notably, it can accept *ICommonFeedbackProvider* (single object) and *IDMFeedbackProvider* (objects linked to decision makers) objects. Both are responsible for creating the feedback (*IPreferenceInformation*) given the constructed reference sets. However, the first is dedicated to creating so-called common feedback, i.e., common to all decision makers. This functionality is related to group decision-making and has not yet been implemented. Therefore, the code prohibits providing any implementation of *ICommonFeedbackProvider*. In turn, *IDMFeedbackProvider* is decision-maker-dedicated. One may provide multiple such objects – each linked to a different decision maker (mapping is done via

the hash map). Note that there is a simple static method "*getForSingleDM*" that allows constructing a suitably parameterized params container (for a single decision maker).

Listing 25: FeedbackProvider class

```
/**
 * Higher-level class that wraps all
    feedback providers ({@link
    IDMFeedbackProvider}).
 *
 * @author MTomczyk
 */
public class FeedbackProvider
{
  /**
   * Params container.
   */
  protected static class Params
  {
    /**
    * Feedback provider that is common to
       all decision makers (either common
       or per-dm providers should be used,
       not both).
    */
    private ICommonFeedbackProvider
       _commonFeedbackProvider;

    /**
     * Feedback providers that are unique
        to each decision maker (accessed
        via decision maker's string
        representation,
     * when linked to a decision maker,
        either common or per-dm providers
        should be used, not both).
     */
    private HashMap<String,
       IDMFeedbackProvider>
       _dmFeedbackProviders;

    [...]

    /**
     * Creates default object instance (
        default parameterization) that uses
         a single feedback provider
        associated
     * with a single decision maker.
     *
     * @param dm       decision maker's
        string representation
     * @param provider common feedback
        provider
     * @return default object instance
     */
    public static Params getForSingleDM(
       String dm, IDMFeedbackProvider
       provider)
    {
      Params p = new Params();
      p._commonFeedbackProvider = null;
      p._dmFeedbackProviders = new HashMap
          <>();
      p._dmFeedbackProviders.put(dm,
          provider);
      return p;
    }

  }

  [...]
}
```

The *IDMFeedbackProvider* is provided in Listing 26 for convenience (its most relevant parts). The central method is here "*getFeedback*". Its implementations are supposed to construct the preference information (*IPreferenceInformation*) wrapper using an auxiliary *DMResult* class, given the decision-maker-related input (decision-maker-dedicated reference sets wrapped using the *ReferenceSets* class).

Listing 26: IDMFeedbackProvider interface

```
/**
 * Interface for classes responsible for
    constructing feedback ({@link
    preference.IPreferenceInformation})
    using the
 * constructed reference sets (each object
    is supposed to be linked with a single
    decision maker).
 *
 * @author MTomczyk
 */
public interface IDMFeedbackProvider
{
  [...]

  /**
   * The main method for providing the
      feedback (wrapped via {@link DMResult
      }).
   *
   * @param referenceSets reference sets
      that are the base for evaluation
   * @return feedback (wrapped via {@link
      DMResult}).
   * @throws FeedbackProviderException the
      exception can be thrown and
      propagated higher
   */
  DMResult getFeedback(ReferenceSets
      referenceSets) throws
      FeedbackProviderException;
}
```

When it comes to concrete implementations, JECDM initially delivers a so-called artificial decision maker based on a value model: *ArtificialValueDM* (see the *interaction.feedbackprovider.dm.artificial.value* package). The artificial decision maker will simulate the answers of a real decision maker using an internal pre-fixed preference model. That is,

the input alternatives (stored) in the reference sets will be evaluated using the model, and relevant conclusions (*IPreferenceInformation*) will be made. Note that, however, the construction of concrete preference information objects is delegated to auxiliary objects implementing the *IFormConstructor* interface (see Listing 27). The available *PairwiseComparisons* implementation will translate the input reference sets into PairwiseComparison objects (*preference.indirect*). Finally, Listing 28 provides the most relevant code snippets of the *ArtificialValueDM* implementation.

Listing 27: IFormConstructor interface

```
/**
 * Interface for classes responsible for
    providing concrete forms of preference
    information given the input reference
    sets
 * (associated with {@link
    ArtificialValueDM}).
 *
 * @author MTomczyk
 */
public interface IFormConstructor<T extends
    AbstractValueInternalModel>
{
  /**
   * Signature for the method responsible
     for retrieving pieces of preferences
     having a specified form.
   *
   * @param referenceSets input reference
     sets derived via reference sets
     constructor ({@link interaction.
     reference.ReferenceSetsConstructor})
   * @param model          preference model
     associated with the artificial
     decision maker ({@link
     ArtificialValueDM})
   * @return preference information
     retrieved
   * @throws FeedbackProviderException the
     exception can be thrown and
     propagated higher
   */
  LinkedList<IPreferenceInformation>
     getFeedback(ReferenceSets
     referenceSets, IPreferenceModel<T>
     model) throws
     FeedbackProviderException;
}
```

Listing 28: ArtificialValueDM class

```
/**
 * Implementation of an artificial DM for
    providing a feedback. It is assumed
    that the internal model employed for
 * simulating the evaluation is a value
    model.
 *
 * @author MTomczyk
 */
public class ArtificialValueDM<T extends
    AbstractValueInternalModel> extends
    AbstractDMFeedbackProvider implements
    IDMFeedbackProvider
{
  /**
   * Preference model (value model) used to
       simulate the decision maker's
      preferences.
   */
  private final IPreferenceModel<T> _model;

  /**
   * Form constructors used by the
      artificial decision maker. These
      objects transform reference sets into
   * concrete answers (e.g., pairwise
      comparisons).
   */
  private final LinkedList<IFormConstructor
     <T>> _formConstructors;

  [...]

  /**
   * The main method for providing the
      feedback (wrapped via {@link DMResult
      }).
   *
   * @param referenceSets reference sets
      that are the base for evaluation
   * @return feedback (wrapped via {@link
      DMResult}).
   * @throws FeedbackProviderException the
      exception can be thrown and
      propagated higher
   */
  @Override
  public DMResult getFeedback(ReferenceSets
      referenceSets) throws
     FeedbackProviderException
  {
    if (referenceSets == null)
      throw new FeedbackProviderException("
         The reference sets are not
         provided (the input is null)",
         this.getClass());

    DMResult result = new DMResult(
       _dmContext, _dm);
    long startTime = System.nanoTime();
    result._feedback = new LinkedList<>();

    for (IFormConstructor<T> fc :
       _formConstructors)
    {
      LinkedList<IPreferenceInformation> pi
         = fc.getFeedback(referenceSets,
         _model);
      if ((pi != null) && (!pi.isEmpty()))
         result._feedback.addAll(pi);
```

```
        }

        result._processingTime = (System.
            nanoTime() - startTime) / 1000000;
        return result;
    }
}
```

**Tutorial4a** source exemplifies how to instantiate an artificial decision maker and use it to obtain the feedback (as with the preceding tutorials, this code mimics the flow imposed by the implemented *PreferenceElicitationModule* – so the user will not have to execute all these lines of code manually in own implementations). The considered scenario is as follows:

- There are two cost-type criteria.
- The decision maker's preferences are simulated using $L_\infty^{[0.7,0.3]}$-norm (so the first criterion is more important to the decision maker than the second).
- The objective space bounds are $[0, 2]^2$ (the artificial decision maker's internal model is already supplied with proper normalization functions).
- The decision maker (artificial) will compare two alternatives: $A0 = [2, 0]$ and $A1 = [0, 2]$. Hence, the *PairwiseComparisons* (*IFormConstructor*) is employed.

Given the above conditions, it is evident that the decision maker should prefer $A1$ over $A0$. It is up to the reader to get acquainted with the tutorial's source code (the code is straightforward). However, Figure 21 illustrated the expected console output for convenience – and it confirms that $A1$ is more relevant to the artificial decision maker than $A0$.

```
Feedback provided by decision maker = DM:
  Iteration = 0
  Date and time (decision-making context) = [6.12.2024 12:51:6:902]
  Date and time (report creation) = [6.12.2024 12:51:6:905]
  Retrieving time = 8 ms
  Feedback:
    Pairwise comparison: A1 PREFERENCE A0
```

Figure 21: Expected console output

**Tutorial4b** differs from Tutorial4a by:

- employing two decision makers modeled with $L_\infty^{[0.7,0.3]}$-norm and $L_\infty^{[0.3,0.7]}$-norm;
- using *FeedbackProvider* to create the feedback.

Consequently, the first decision maker should still prefer $A1$ over $A0$, but the preference should be inversed for the second decision maker. Note that the code uses a not-yet-introduced class: DM (decision maker's identifier) to properly utilize *FeedbackProvider*. This class will be overviewed later. Figure 22 showcases the expected console output – the log on feedback-providing process. The log confirms the expectations regarding how different decision makers will compare presented alternatives.

```
Iteration = 0
Date and time (decision-making context) = [6.12.2024 12:51:32:981]
Date and time (report creation) = [6.12.2024 12:51:32:982]
Retrieving time = 11 ms
Feedback retrieved:
  Feedback provided by decision maker = DM1:
    Iteration = 0
    Date and time (decision-making context) = [6.12.2024 12:51:32:981]
    Date and time (report creation) = [6.12.2024 12:51:32:982]
    Retrieving time = 9 ms
    Feedback:
      Pairwise comparison: A1 PREFERENCE A0
  Feedback provided by decision maker = DM2:
    Iteration = 0
    Date and time (decision-making context) = [6.12.2024 12:51:32:981]
    Date and time (report creation) = [6.12.2024 12:51:32:997]
    Retrieving time = 0 ms
    Feedback:
      Pairwise comparison: A0 PREFERENCE A1
```

Figure 22: Expected console output

### 3.5. Complex example [T]

Source package: *t1_t10.t4_decision_support_module.* ***t2_preference_elicitation_module.t5_complex_example***

The last tutorial on executing preference elicitation concerns creating a *PreferenceElicitationModule* and coupling it with the MODE/A algorithm. Note that the case is highly artificial, which means it concerns just using the module, not presenting any practical – and reasonable – application. Further, the implementation is relatively hard-coded. Nonetheless, it intends to exemplify how the relevant system's components operate. Therefore, these flaws are acceptable. Note that *PreferenceElicitationModule* is not supposed to be constructed manually by the used in typical applications – it is the responsibility of the highest-level *DecisionSupportSystem*. The scenario this tutorial considers is as follows:

- It applies the MOEA/D algorithm to a 3-dimensional DTLZ2 problem.
- The method is run for 500 generations.
- The preference elicitation module is established for two decision makers: "DM1" and "DM2".
- The module uses *InteractionInterval* as an interaction trigger (every 50 generations, starting from 0), default refiner, *RandomPairs* (with required spread of 0.01) for constructing reference sets, and two artificial decision makers (linked "DM1" and "DM2") for providing the feedback.
- The artificial decision makers are modeled with $L_\infty^{\boldsymbol{w}}$-norms, where $\boldsymbol{w} = [0.7, 0.2, 0.1]$ for "DM1" and $\boldsymbol{w} = [0.2, 0.3, 0.4]$ for "DM2".
- The preference elicitation module is implicitly coupled with *Runner* used to run MOEA/D. The subsequent generation numbers and current populations are the basis for

creating the decision-making context for the preference elicitation module. In other words, the module is fed with populations constructed by MOEA/D. The preference elicitation is triggered if the generation number satisfies the interaction trigger. Overall, the *PreferenceElicitationModule.executeProcess* method is called to execute the preference elicitation process.

- Plot3D is used to visualize the simulation. The population constructed by MOEA/D is drawn using regular markers (gradient color based on Z-dimension). Additionally, the solutions compared pairwise by the decision makers are illustrated using line segments ending with markers. These markers are filled with different colors based on the preference direction (red = not preferred; green/blue = preferred to "DM1/2".

No codes are brought here as they present barely anything new. Figure 23 illustrates the expected visualization outcome (after all generations are passed). At the same time, Figure 28 depicts the partial console output (part of the report on the most recent preference elicitations and the string representation of the preference elicitation history for "DM1". Note that the pairs to be compared by the decision makers were selected randomly. Hence, the illustration may seem chaotic at first. Nonetheless, this example illustrates well how a preference elicitation can be coupled with an evolutionary algorithm to build a preference elicitation history (the details of this process can be found in the printed log). Additionally, observe that the solutions preferred by "DM1" (in pairs; green pyramids) appear more frequently for lower values of the first objective function. It aligns with how the first artificial decision maker was formulated (the first weight for the $L_\alpha^w$-norm was the most significant). Similarly, the preferred alternatives for "DM2" tend to minimize the third objective (blue cubes), which also matches the description of the preference model used to simulate the second decision maker's answers.

Finally, Figures 24-27 illustrate expected outcomes when using slightly different settings. First, the scenario in Figure 24 assumed 10 times more frequent interactions (after every 5 generations). Next, Figure 25 is based on the initial assumption regarding the interaction interval (after every 50 generations) but uses an increased required spread threshold 0.01 → 0.2. This change neglected a selection of those alternatives that were too close to one another to form a pair. However, the impact of using greater thresholds is more clearly visible in Figure 26 (threshold of 0.5) and 27 (threshold of 1.0). The greater the threshold, the greater the distances between the alternatives selected to form pairs. If the threshold is significantly large, reference sets may not even be constructed in interaction rounds.

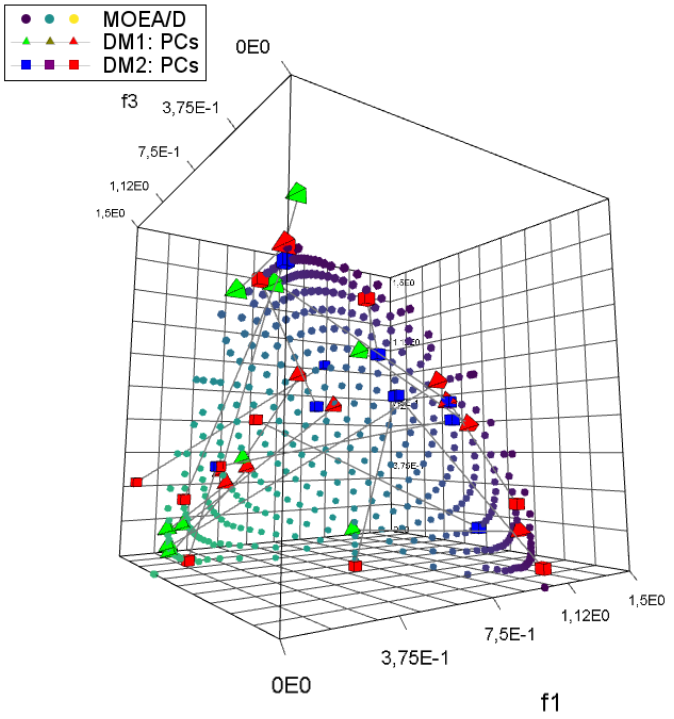



Figure 23: Expected final visualization (10 interactions for each decision maker; *threshold* = 0.01)

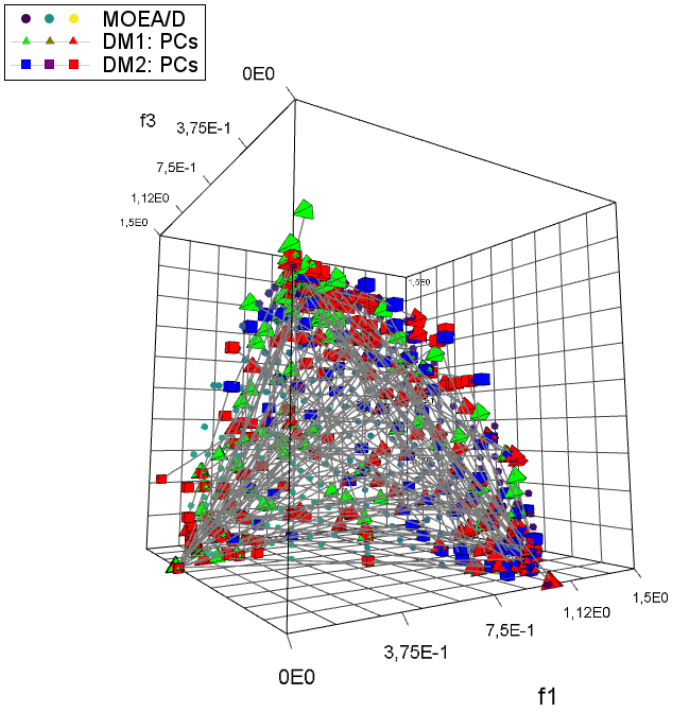



Figure 24: Expected final visualization (100 interactions for each decision maker; *threshold* = 0.01)

## 4. Updating preference models

This framework is centrally focused on implementing methods incorporating the **preference learning** paradigm [3]. Preference learning is about learning how to describe the decision maker's internal preference system mathematically. The learning process should rely on the feedback provided by them. Naturally, a method has to be supplied with a model that has to be accordingly tuned and exploited. In some sense, this is a ma-

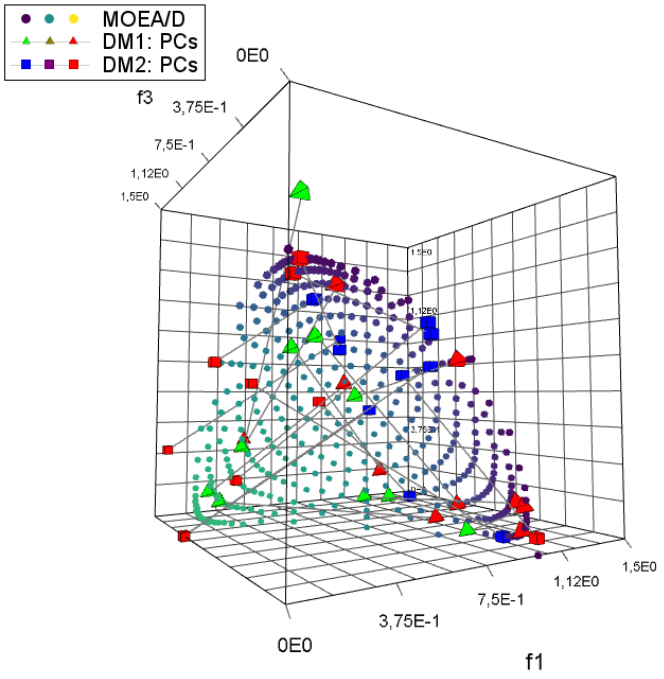



Figure 25: Expected final visualization (10 interactions for each decision maker; *threshold* = 0.2)

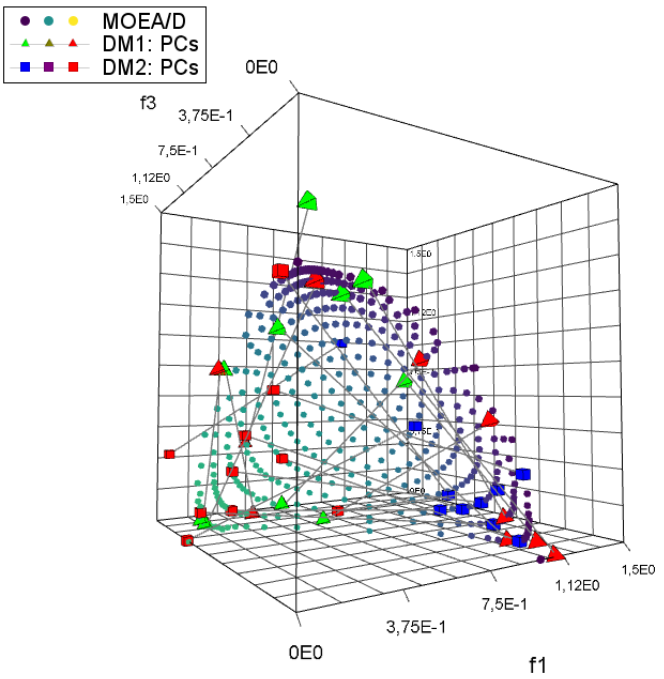



Figure 26: Expected final visualization (10 interactions for each decision maker; *threshold* = 0.5)

chine learning task. However, there are some differences between preference learning and machine learning, which are related to assumptions made. First, preference learning avoids using black box models – e.g., neural networks – to ensure interpretability. Second, it aims to minimize the decision maker's

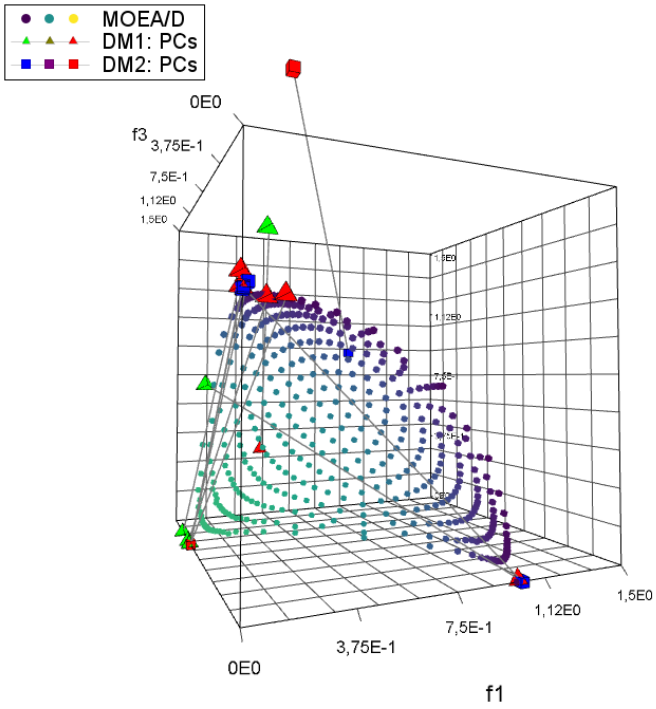



Figure 27: Expected final visualization (10 interactions for each decision maker; *threshold* = 1.0)

effort needed to be spent to understand their internal value system adequately. It means that the system should avoid questioning them (i) too often (e.g., few interactions are preferred) and (ii) too intensively (too technical questions should be avoided). It contrasts the typical characteristics of training data encountered in machine learning.

The initial release of JECDM focuses on so-called value models (primarily $L_\alpha^{\boldsymbol{w}}$-norms) and pairwise comparisons (feedback form). Given these two concepts, the fundamentals of preference learning are briefly described below. Naturally, the framework is open to new models/preference forms, which will be added in future releases.

*Preference model.* First, let us bring the already introduced in Section 3 definition of the $L_\alpha^{\boldsymbol{w}}$-norm model:

$$L_\alpha^{\boldsymbol{w}}(\boldsymbol{s}) = \left[\sum_{i=1}^{M} (s_i w_i)^\alpha\right]^{\frac{1}{\alpha}}. \tag{2}$$

For the sake of simplicity, assume that all alternatives are already normalized – so standardization is implicitly applied. By using this $L_\alpha^{\boldsymbol{w}}$-norm as the preference model, the method's designer assumes that it can **well represent/simulate the decision maker's preference system**. The "represent" term refers to parameterization. Initially, when no feedback is collected, any model instance can be called "compatible". However, in subsequent interactions with the decision maker, additional data can be gathered, making some model instances more or less suited to how (s)he expressed their aspirations. Specifically, assume that $\mathcal{H}$ represents a history of preference elicitation, and it is

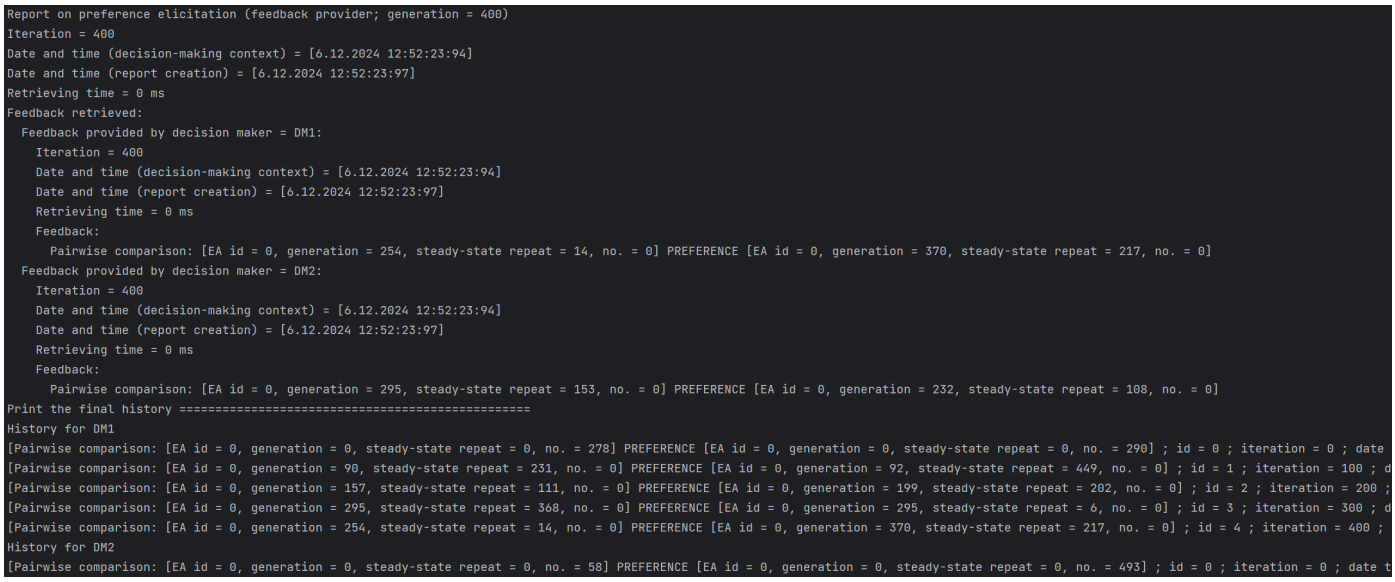



Figure 28: Expected console output (10 interactions for each decision maker)

composed of $k$ compared by the decision maker performance vectors, i.e.,

$$\mathcal{H} = \left\{ \left(\boldsymbol{s}_1^A \succ \boldsymbol{s}_1^B\right)_1, \left(\boldsymbol{s}_2^A \succ \boldsymbol{s}_2^B\right)_2, ..., \left(\boldsymbol{s}_k^A \succ \boldsymbol{s}_k^B\right)_k \right\},$$

where $\succ$ denotes the preference relation (the left alternative is preferred over the right).

*Compatible model instances.* Intuitively, we can can say that a model instance – $L_\alpha^{\boldsymbol{w}}$-norm with some provided compensation level $\alpha$ and weight vector $\boldsymbol{w}$ – is compatible with a pairwise comparison $\left(\boldsymbol{s}^A \succ \boldsymbol{s}^B\right)$ **if it ranks $\boldsymbol{s}^A$ better than $\boldsymbol{s}^B$**, i.e., $L_\alpha^{\boldsymbol{w}}(\boldsymbol{s}^A) < L_\alpha^{\boldsymbol{w}}(\boldsymbol{s}^B)$. If the model instance satisfies the above conditions for all feedback stored in $\mathcal{H}$, we call such model instance – and its underlying parameters – **compatible with the decision maker's preference information**. Formally:

$$\text{iff } \forall_{\left(\boldsymbol{s}^A \succ \boldsymbol{s}^B\right) \in \mathcal{H}} L_\alpha^{\boldsymbol{w}}(\boldsymbol{s}^A) < L_\alpha^{\boldsymbol{w}}(\boldsymbol{s}^B) \rightarrow \textit{the model is compatible.} \tag{3}$$

*Exploiting compatible models.* For the sake of simplicity, let us assume now that $\alpha$ is fixed, and only $\boldsymbol{w}$ is considered a model parameter. Naturally, there may be (infinitely) many compatible such models. By $\mathcal{W}$, we will denote a space for all such compatible weight vectors. The model may quantify alternatives entirely differently, given its different compatible instances. Nonetheless, all these instances align with the feedback collected, increasing their credibility regarding their relevance to the decision maker. When it comes to deriving final recommendations, various methods from the stream of multi-criteria decision analysis employ so-called robust procedures that assess alternatives in view of all such compatible models simultaneously (see, e.g., the robust ordinal regression [4] or determining the potential optimality [5]).

*Inconsistency.* The space of compatible model parameters $\mathcal{W}$ may be empty, however. If so, no credible recommendations can be derived – because it remains unknown how to construct a model instance. The inconsistency will happen when (i) the employed model cannot fully represent the decision maker's value system, or they provided contradictory feedback (e.g., due to the irrationality). No matter the cause, **such case will be referred to as inconsistency**, and a suitable procedure for reintroducing consistency will be needed to resolve this issue. The inconsistency handling procedure may be based on different principles. For instance, the repairing procedure implemented in NEMO-II [6] algorithm strategically removes some of the collected feedback until compatibility is restored. In turn, CIEMO/D [7] simultaneously optimizes sub-populations linked to different models and prioritizes those of the populations that remain the most consistent with the decision maker's feedback.

Now, let us focus on how preference learning is implemented in JECDM from the system's perspective. First, all essential processing is handled by the top-level *ModelsUpdatedModule* (see Figure 2; *system.modules* package). As with *PreferenceElicitationModule*, this module imposes a hierarchical processing, and each component provides a report-like object as a return type. These reports are aggregated when moving from the bottom to the top of the class hierarchy, providing, in the end, a comprehensive report on the module's processing. **Important note**: As with *PreferenceElicitationModule*, the following discussion briefly showcases the processing, which is an integral part of the implemented system. Therefore, it is recommended not to modify its code to avoid inconsistency with future releases. The decision support system is expected to be parameterized and run only via *DecisionSupportSystem* class.

Figure 29 provides a (very) simplified scheme of the processing imposed by the central *PreferenceElicitationModule.executeProcess* method. As with *PreferenceElicitationModule*, the process starts with registering the current decision-

making context. Next, it iterates over all *DecisionMakerSystem* objects maintained and calls their "*updateModels*" method (see *system.dm* package). The *DecisionMakerSystem* is just below *DecisionSupportSystem* in the primary class hierarchy – see Figure 1. It is responsible for maintaining all data linked with one pre-determined decision maker. Its three central fields are *DM _DMs* – decision maker's system identifier, *ModelSystem<?>[] _modelSystems* – associated model systems (each can be of a different generic type, but it must be an extension of *AbstractInternalModel*; see *system.model* package), and *History _history* – stores the history of preference elicitation. Each *DecisionMakerSystem* can be supplied with multiple *ModelSystem* objects. As already explained, the decision maker can be associated with multiple – usually different – preference models.

Next, each model system (for each decision maker) is called its own "*updateModels*" method. At this point, it is worth reminding that the results of these calls are returned as report-like objects, which are captured and suitably handled by the method's caller. One model system is responsible for per-model processing. Listing 29 provides its code snippets related to its parameterization. Notably, there are three central parameterization means (provided via params container): *IPreferenceModel<T> _preferenceModel* preference model definition associated with the system, *IConstructor<T> _modelConstructor* – object responsible for learning how to parameterize the preference model (constructs internal models for *_preferenceModel*), and *IInconsistencyHandler<T> _inconsistencyHandler* – object responsible for dealing with inconsistencies. Notably, the model maintains its own "*History _history*". In general, it can be considered a subset (of the history maintained in the higher-level *DecisionMakerSystem* object. As already explained, the model system utilizes its copy. The rationale is to allow the inconsistency handler to freely manipulate this – individual to the model – history when reinstating consistency. The individual, i.e., per-model, histories are updated (new feedback is supplied) mainly during the preference elicitation step via "*notifyAboutMostRecentPreferenceInformation*" calls.

Listing 29: ModelSystem class

```
/**
 * A class representing a model-level
    processing (for one decision maker). It
     is centrally focused on updating the
 * assumed preference model, given the
    preference examples provided by the
    decision maker, and handling potential
    inconsistencies.
 *
 * @author MTomczyk
 */
public class ModelSystem<T extends
    AbstractInternalModel>
{
  /**
   * Params container.
   */
```

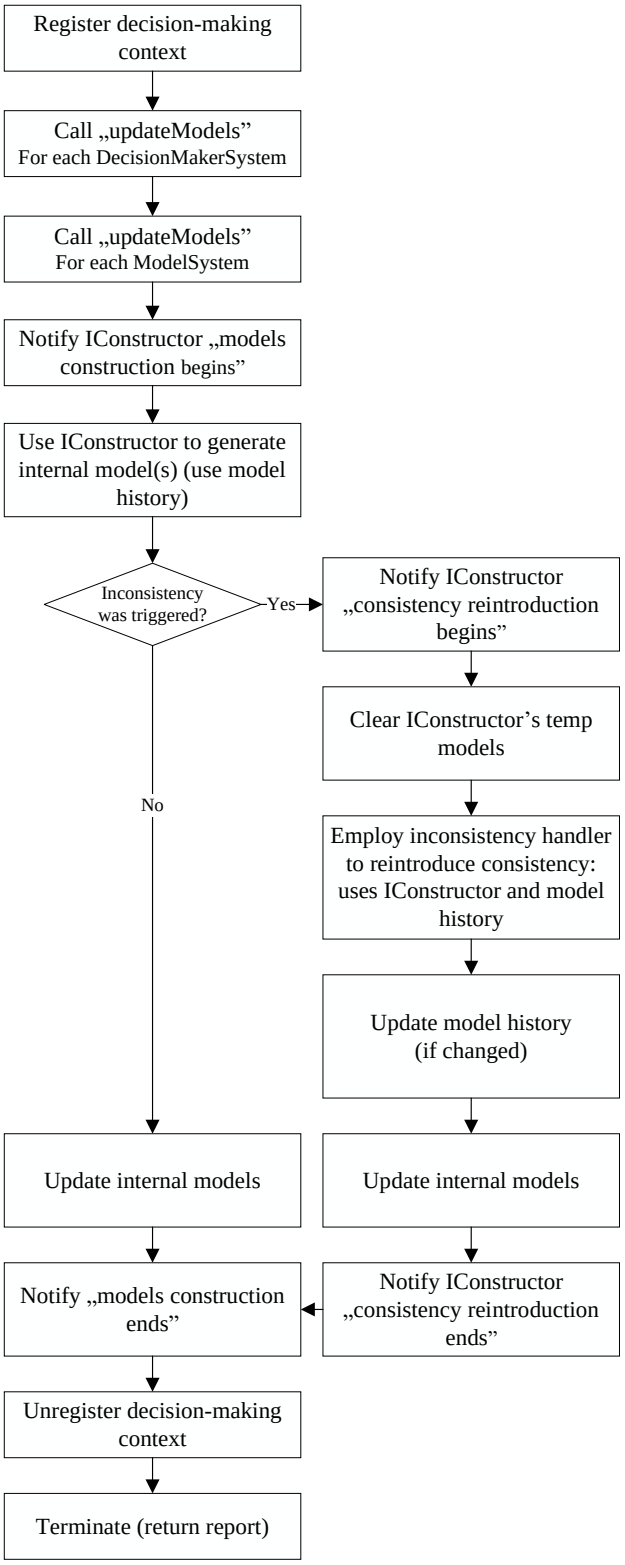



Figure 29: General scheme of the flow imposed by the models updater module.

```
public static class Params<T extends
    AbstractInternalModel>
{
  /**
   * Preference model used to represent
```

```
         the decision maker's preferences.
     */
    public IPreferenceModel<T>
        _preferenceModel;

    /**
     * Preference model constructor.
     */
    public IConstructor<T>
        _modelConstructor;

    /**
     * Object responsible for handling
         inconsistency that can occur when
         building the model(s) being in line
          with the
     * decision maker's aspirations.
       */
    public IInconsistencyHandler<T>
        _inconsistencyHandler;
  }

  /**
   * Preference model used to represent the
       decision maker's preferences.
   */
  private final IPreferenceModel<T>
      _preferenceModel;

  [...]

  /**
   * History of preference elicitation
      associated with the model.
   */
  private final History _history;

  [...]
}
```

The *updateModels* method of *ModelSystem* begins by notifying the *IConstructor* (see *model.constructor* package) object that "models construction begins". Then, the process attempts to learn how to parameterize the preference model. For this purpose, the central method of the *IConstructor* object is called: *Report<T> constructModels(LinkedList<PreferenceInformationWrapper> preferenceInformation) throws ConstructorException;*. This method is responsible for constructing preference model instances (internal models) being compatible with the input feedback (collected from the model history) and returning it via the *Report* object. Listing 30 brings selected code snippets from the *IConstructor* interface for convenience. Note that this interface introduces many system-related methods, e.g., "notify...". They are primarily implemented in the common *AbstractConstructor* class.

Listing 30: IConstructor interface

```
package model.constructor;

import exeption.ConstructorException;
import history.PreferenceInformationWrapper
    ;
import model.internals.
    AbstractInternalModel;
import dmcontext.DMContext;

import java.util.LinkedList;

/**
 * General interface for classes
     responsible for constructing (internal)
      models given the decision maker's
     preference
 * information provided.
 *
 * @author MTomczyk
 */
public interface IConstructor<T extends
    AbstractInternalModel>
{
  /**
   * Auxiliary method that can be called to
       clear all stored models (can be
      called, e.g., when the inconsistency
      was detected).
   */
  void clearModels();

  /**
   * Auxiliary method that can be used to
      register the current decision-making
      context {@link DMContext}.
   *
   * @param dmContext decision-making
      context
   * @throws ConstructorException the
      exception can be thrown and
      propagated higher
   */
  void registerDecisionMakingContext(
      DMContext dmContext) throws
      ConstructorException;

  [...]

  /**
   * The main method for constructing
      internal models (array) given the
      decision maker's preference
      information.
   *
   * @param preferenceInformation the
      decision maker's preference
      information stored (provided via
      wrappers)
   * @return constructed models (bundle
      object that also provides additional
      data)
   * @throws ConstructorException the
      exception can be thrown and
      propagated higher
```

```
     */
    Report<T> constructModels(LinkedList<
        PreferenceInformationWrapper>
        preferenceInformation) throws
        ConstructorException;
    [...]
}
```

If the construction process terminates successfully, i.e., without detecting inconsistency, it updates the preference model's internal models, notifies that "construction ends," unregisters the decision-making context, and terminates by returning a comprehensive report on its execution. Suppose, however, that *IConstructor* cannot build suitable internal models. If so, the process triggers the procedure for reinstating consistency. *Important note*: It is assumed that this procedure will terminate successfully no matter what – even at the cost of completely clearing the individual model's preference elicitation history (no feedback implies that all model instances should be consistent).

Reinstating consistency begins by notifying *IConstructor* that "consistency reintroduction begins. Implementations may use this notification to, e.g., adjust model meta-parameters (e.g., the compensation level $\alpha$ for $L^{w}_{\alpha}$-norms hoping that the constructor will be able to create compatible with such adjusted meta-parameter). Next, the *IConstructor.clearModels()* method is called to clear the constructor's cached models (constructors may cache their constructed models). Next the inconsistency handler is executed (*IInconsistencyHandler*). It is supplied with three core inputs: report on the initial construction process (report generated by *IConstructor* in the first construction attempt – may be used by the handler to suitably adjust processing), the constructor itself (the handler is supposed to utilize the constructor to create compatible model instances), and the model's individual preference elicitation history – represented as list of *PreferenceInformationWrapper* objects (ordered from the oldest to the newest, derived via *history.getPreferenceInformationCopy()*).

The inconsistency handler is expected to rerun *IConstruct* object given different conditions until the compatibility is restored. The term "conditions" may refer, e.g., to model-meta parameters (changing them and seeing what will happen) or altering the model's feedback collected. The latter is the foundation of the *RemoveOldest* implementation (see *inconsistency* package), which was inspired by the inconsistency handling technique presented in [6]. This procedure iteratively removes the collected feedback, starting from the oldest, until the feasibility is restored. Then, it tries to iteratively reintroduce as many of the removed preference statements (starting from the newest) without violating compatibility. The procedure will be examined in later sections as a tutorial example. For interested readers: each "condition" is represented as *State* class (can be overwritten for customization), and it can be provided with the resulting report on consistency reintroduction.

Listing 31: IInconsistencyHandler interface

```
/**
 * Interface for classes responsible for
    handling inconsistency when
    constructing the model given preference
     information
 * (reintroduce consistency).
 *
 * @author MTomczyk
 */
public interface IInconsistencyHandler<T
    extends AbstractInternalModel>
{
  /**
   * Signature for the method responsible
      for reintroducing consistency.
       *
   * @param report bundle result of the
      attempt to construct models (see {
      @link Report},
   *                    it is assumed that
      this bundle indicates inconsistency
      ({@link Report#_inconsistencyDetected
      }).
   * @param constructor object used to
      construct the model given the
      preference information stored in the
      history object {see {@link History}).
   * @param preferenceInformation current
      preference information (copied from
      the model system) that can be
      modified to reintroduce consistency (
      altered set should be stored in the
      report);
   * the list is derived via {@link History
      #getPreferenceInformationCopy()},
      thus it is valid (e.g., preference
      statements are ordered from the
      oldest to the newest)
   * @return the method should reintroduce
      consistency (e.g., by altering the
      constructor definition or history
   * of preference elicitation); the result
       should be provided via the report
      object
   * @throws InconsistencyHandlerException
      the preference model exception can be
       thrown and propagated higher
   */
  inconsistency.Report<T>
      reintroduceConsistency(Report<T>
      report, IConstructor<T> constructor,
      History history) throws
      InconsistencyHandlerException;
}
```

After consistent model(s) are found, the model history is updated (replaced with the returned by *IInconsistencyHandler* altered set), the generated compatible model instances are stored as the preference model's internal ones, and the *IConstructor* is notified that "consistency reintroduction ends". Finally, the process returns to the regular termination path: notify that "models construction ends", unregisters the decision-making context,

and return the report on the whole process.

In what follows, the document delves into individual components of the above-discussed processing and provides relevant tutorial examples.

### *4.1. Constructing compatible model instances [T]*

Source package: *t1_t10.t4_decision_support_module.*
***t3_models_updater_module.***
***t1_constructing_compatible_models***

The initial release of JECDM supplies one central implementation of the *IConstruct* interface: *FRS* (see *model.constructor.value.frs* package; note that there are also some auxiliary, algorithm-dependent implementations). FRS stands for "fast rejection sampling". Is it a procedure based on the Monte Carlo simulation [7, 8]. Its underlying idea is straightforward – it generates random model instances given the feedback used to constrain model parameter space. The process continues until a desired number of samples (models) is identified. However, if the required minimal number of samples is not found (0, by default), the procedure returns a report stating that the inconsistency occurred.

Listing 32 brings *FRS.Params* container for convenience. Let's discuss its parameterization means. First, note that *FRS* (and its params container) are generic classes, and the parameter type is bounded from above by *AbstractValueInternalModel*. It means that this constructor can be parameterized to construct any model instances as long as these are value models (of the same type).

Regarding the parameters, *IRandomModel<T> _RM* is an object to which the task of generating random model instances will be delegated (see the model.constructor.random package). *LNormGenerator* is a dedicated implementation for sampling $L_\alpha^{\mathbf{w}}$-norms. Next, a normalization builder used when standardizing the performance vector can be provided – it is automatically set to *StandardLinearBuilder* by default.

The following three integer parameters can be considered fundamental. The "*int _sampligLimit*" limits the number of samples the constructor is allowed to generate. Next, "int *_feasibleSamplesToGenerate*" specifies the requested number of compatible model instances to generate("*int _sampligLimit*" should not be smaller than "int *_feasibleSamplesToGenerate*"). Naturally, the constructor will immediately terminate sampling if the required number of models is derived. Finally, *int _inconsistencyThreshold* determines the maximum number of compatible models found that still make the final result inconsistent. By default, this value is set to 0, meaning that the inconsistency will reported only when no compatible models are found.

Next, the "*boolean _validateAlreadyExistingSamplesFirst*" flag – it is set to true by default – dictates whether to check already cached models (obtained in previous calls) before sampling new ones. The rationale is that many may preserve consistency after receiving new feedback and, subsequentially, calling the constructor again. Hence, they may be preserved, giving the sampler better starting conditions. Finally, "*CompatibilityAnalyzer _validateAlreadyExistingSamplesFirst*" is an auxiliary object (instantiated by default) that provides a means for verifying the compatibility given the decision maker's feedback. Note that the current implementation is focused primarily on pairwise comparisons. However, its functionalities will be extended in future releases.

Listing 32: FRS.Params class

```
/**
 * Params container.
 */
public static class Params<T extends
    AbstractValueInternalModel>
{
  /**
   * Random model generator.
   */
  public final IRandomModel<T> _RM;

  /**
   * Auxiliary normalization builder object
       that can be used to generate
      normalization objects.
   * By default, standard linear builder is
       used.
   */
  public INormalizationBuilder
      _normalizationBuilder;

  /**
   * Limit for the number of samples the
      FRS method is allowed to generate.
      Should be positive and greater (or
      equal)
   * than {@link Params#
      _feasibleSamplesToGenerate}. In the
      case of value violation, it is set to
       1 or
   * {@link Params#
      _feasibleSamplesToGenerate} (whatever
       is greater).
   */
  public int _samplingLimit = 1000000;

  /**
   * The number of feasible samples the
      method is supposed to generate. This
      number should be positive.
   * If it is not, it is set to 1.
   */
  public int _feasibleSamplesToGenerate =
      1;

  /**
   * The threshold for the number of
      generated compatible samples. The
      overall processing result is
      considered
   * inconsistent if the number is smaller
      or equal to this threshold.
```

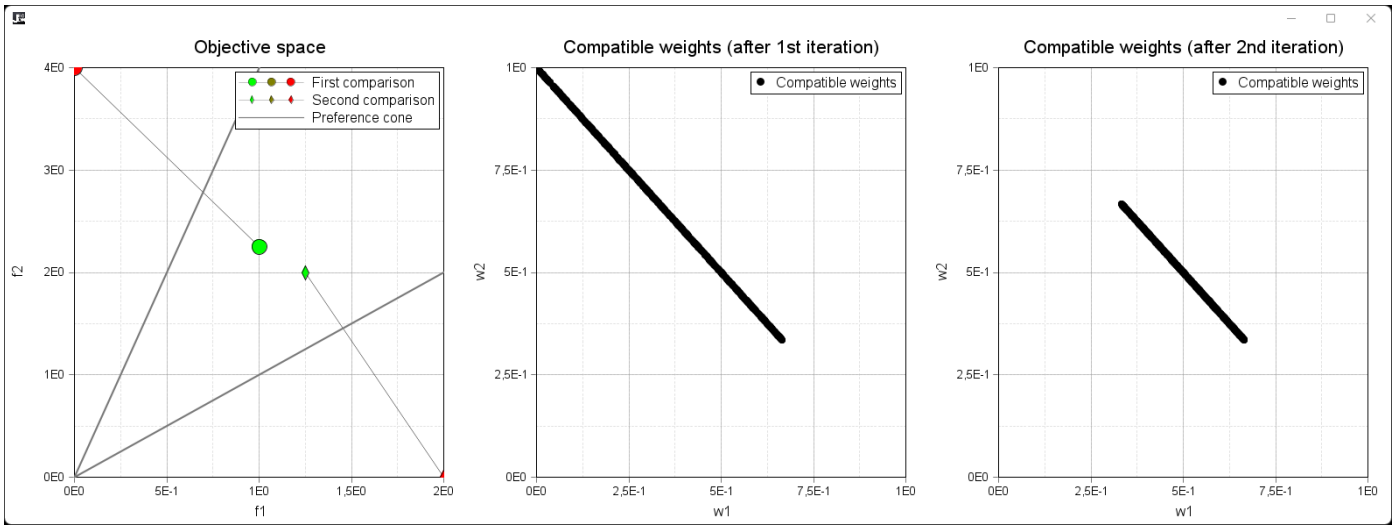



Figure 30: Expected final visualization (2D case)

```
    */
   public int _inconsistencyThreshold = 0;

   /**
    * If true, the FRS method first checks
        the already existing samples and
        preserves those that are compatible
    * with the current preference
        information set. When false, FRS
        generates everything from scratch.
    */
   public boolean
       _validateAlreadyExistingSamplesFirst =
        true;

   /**
    * Auxiliary object responsible for
        calculating the degree to which a
        given preference information is
        compatible
    * with an input preference model.
    */
   public CompatibilityAnalyzer
       _compatibilityAnalyzer = new
       CompatibilityAnalyzer();

   [...]
}
```

**Tutorial1a** source code showcases how to instantiate and use *FRS* in practice. Again, the underlying scenario is highly artificial, and – in typical applications, the execution of *FRS* is handled by *ModelSystem*. The setting is as follows:

- Two cost-type criteria are considered, and the known bounds for objective space are $[0.0, 2.0]$ and $[0.0, 4.0]$.
- The feedback consists of two pairwise comparisons: $[1.0, 2.25] \succ [0.0, 4.0]$ and $[1.25, 2.0] \succ [2.0, 0.0]$ (coordinates are in the objective space). These preference statements are supplied sequentially in two iteration rounds.

```
Iteration = 0 ========================
History (tutorial)
[Pairwise comparison: A0 PREFERENCE A1 ; id = 0 ; iteration = 0 ;
Iteration = 0
Date and time (decision-making context) = [13.11.2024 15:8:7:324]
Date and time (report creation) = [13.11.2024 15:8:7:324]
Inconsistency detected = false
The number of generated models = 1000
Elapsed time = 7 ms
Success rate in preserving = 0.0
Models preserved between iterations = 0
Models rejected between iterations = 0
Success rate in constructing = 0.6544502617801047
Accepted newly constructed models = 1000
Rejected newly constructed models = 528
Normalizations were updated = true

Iteration = 1 ========================
History (tutorial)
[Pairwise comparison: A0 PREFERENCE A1 ; id = 0 ; iteration = 0 ;
[Pairwise comparison: A2 PREFERENCE A3 ; id = 1 ; iteration = 1 ;
Iteration = 1
Date and time (decision-making context) = [13.11.2024 15:8:7:344]
Date and time (report creation) = [13.11.2024 15:8:7:345]
Inconsistency detected = false
The number of generated models = 1000
Elapsed time = 2 ms
Success rate in preserving = 0.493
Models preserved between iterations = 493
Models rejected between iterations = 507
Success rate in constructing = 0.3418745785569791
Accepted newly constructed models = 507
Rejected newly constructed models = 976
Normalizations were updated = false
```

Figure 31: Expected console output (2D case)

- The *FRS* is set to generate 1000 compatible model instances (limit = 1000000; inconsistency threshold = 0). The *LNormGenerator* is employed to generate samples (the $\alpha$ parameter is fixed and set to $\infty$).

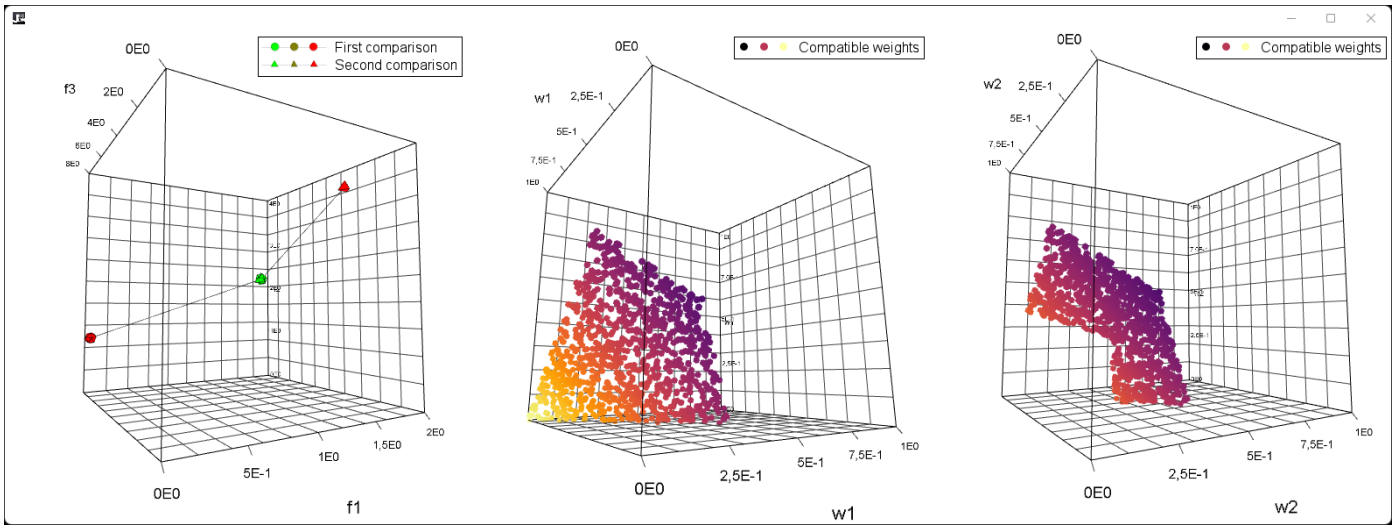



Figure 32: Expected final visualization (3D case)

- The initial scenario parameters are set so that no inconsistency should be detected.

The outcomes will be presented twofold. First, the reports on model construction will be logged in the console. Second, a visualization module will be employed to depict the outcomes. Specifically, a frame consisting of three plots will be displayed. The first plot is set to illustrate the pairwise comparisons (objective space; preferred alternatives are marked using green marker fill color). The second and the third plots are set to illustrate compatible weight vectors of sampled models (hence, they depict the model parameter space).

Figure 30 illustrates the final result (visualization). Evidently, pairwise comparisons in this scenario favor balanced weight vectors more than extreme ones. **Important reminder:** It is assumed that all relevant calculations are done in the normalized space. Hence, the compared alternatives are suitably rescaled when inspecting the compatibility of a candidate preference model instance.

Note that it is relatively easy to determine feasible weights analytically if the number of objectives is two and the underlying model is $L^{\boldsymbol{w}}_{\infty}$-norm = Chebyshev function. Let me refer to [9] for details. In short, the pairwise comparisons form a so-called preference cone (illustrated in the plot). Objective vectors between these two boundary lines are preferred over alternatives being "outside". Next, a weight vector can be transformed in a (normalized) direction of such a line by calculating its inverse values and performing normalization. Regarding the first pairwise comparison, intuitively, all such boundaries whose directions are "below" the boundary line imposed by the the feedback are compatible. Thus, one may start by stating that the $f_1$-coordinate of such normalized lines should be $\geq 1/3$ (can be easily read from the left plot in Figure 30). Then, one can transform the first weight vector $w_1$ into the first normalized direction coordinate, imposing also the restriction on its lower bound: $w/w_1/(1/w_1 + 1/w_2) \geq 1/3 \rightarrow 1/w_1 \geq 1/3(1/w_1 + 1/w_2) \rightarrow w_2 \geq 1/3(w_1 + w_2) \rightarrow w_2 \geq 1/3$. Thus, all compatible weight vectors can be expressed as $\{[w_1, w_2 = 1 - w_1] : w1 \in [0, 2/3]\}$. It finds confirmation in the sampled set illustrated in the central plot in Figure 30. Further, the compatible vectors constitute $2/3$ of all possible vectors, meaning that the success rate of *FRS* should be around 66.66%. It aligns with the report on the first model construction attempt printed as the console log – see Figure 31 (see iteration 0; success rate in constructing).

Consider then the following iteration. This time, the pairwise comparisons impose the upper limit for $w_2$ of $2/3$. Combined with the previous statement, this yields a set of compatible vectors of $\{[w_1, w_2 = 1 - w_1] : w_1 \in [1/3, 2/3]\}$. Thus, the success rate in constructing compatible vectors should be around 33.33% – which again finds confirmation in the log in Figure 31 (iteration 1). Note that, however, *FRS* was set to inspect the cached model first. Intuitively, (around) half of them should preserve compatibility. Indeed, the log informs that the success rate in preserving is 0.493 (very close to 50%), and 493 models preserved compatibility. It means that just $1000 - 493$ had to be sampled.

Regarding **Tutorial1b** source code, it differs from **Tutorial1a** by considering a three-dimensional objective/performance space. The other settings remain similar. Figures 32 and 33 illustrate the expected visualization and console outcomes. This time, analytically calculating the expected boundaries for weight vectors is slightly more complicated; hence, it is omitted for clarity. As for the rendered compatible weight vectors, they were colored using a gradient based on the Z-axis. Notably, these vectors can now form non-convex structures (see the right plot in Figure 32; these structures are always convex for two-dimensional settings – they constitute line segments). Regarding the efficiency of generating compatible weight vectors, it dropped – which was expected. Moving to more dimensional spaces reduces the chances of sampling compatible model instances.

```
Iteration = 0 ======================
History (tutorial)
[Pairwise comparison: A0 PREFERENCE A1 ; id = 0 ; iteration
Iteration = 0
Date and time (decision-making context) = [13.11.2024 15:9:
Date and time (report creation) = [13.11.2024 15:9:22:303]
Inconsistency detected = false
The number of generated models = 1000
Elapsed time = 8 ms
Success rate in preserving = 0.0
Models preserved between iterations = 0
Models rejected between iterations = 0
Success rate in constructing = 0.45599635202918376
Accepted newly constructed models = 1000
Rejected newly constructed models = 1193
Normalizations were updated = true

Iteration = 1 ======================
History (tutorial)
[Pairwise comparison: A0 PREFERENCE A1 ; id = 0 ; iteration
[Pairwise comparison: A2 PREFERENCE A3 ; id = 1 ; iteration
Iteration = 1
Date and time (decision-making context) = [13.11.2024 15:9:
Date and time (report creation) = [13.11.2024 15:9:22:323]
Inconsistency detected = false
The number of generated models = 1000
Elapsed time = 2 ms
Success rate in preserving = 0.584
Models preserved between iterations = 584
Models rejected between iterations = 416
Success rate in constructing = 0.26229508196721313
Accepted newly constructed models = 416
Rejected newly constructed models = 1170
Normalizations were updated = false
```

Figure 33: Expected console output (3D case)

### 4.2. *Handling inconsistencies [T]*

Source package: *t1_t10.t4_decision_support_module.*
***t3_models_updater_module.***
***t2_hadling_inconsistencies***

This tutorial can be deemed a continuation of the previous one. They differ only in that this tutorial employs 5 pairwise comparisons (5 iterations), uses the *RemoveOldest* inconsistency handler, and does not employ visualization. It intends to showcase how *RemoveOldest* deals with inconsistency. For this purpose, the pairwise comparisons used were intentionally designed so that some of them will certainly – and individually (e.g., independently to other answers) – trigger incompatibility.

Consider Table 4. It showcases all the preference statements employed in this tutorial's source code (performance vectors of (not) preferred alternatives). Some of them are intentionally designed to trigger inconsistency. These are answers no. 1, 3, and 5. Since both criteria employed are of cost-type, these pairwise comparisons state that the decision maker prefers the dominated solution over non-dominated – which cannot be handled by the $L^w_\alpha$-norm (and one cannot expect from a model to capture such statements; such decisions are called irrational). However, such irrational statements allow for simulating inconsistency on demand. Also, note that statements 2 and 4 are simultaneously consistent with the model (the pairwise comparisons were used in the previous example).

| Iteration | Preferred | Not preferred |
|---|---|---|
| 1* | [1.0, 1.0] | [0.0, 0.0] |
| 2 | [1.0, 2.25] | [0.0, 4.0] |
| 3* | [1.0, 1.0] | [0.0, 0.0] |
| 4 | [1.25, 2.0] | [2.0, 0.0] |
| 5* | [1.0, 1.0] | [0.0, 0.0] |

Table 4: Decision maker's feedback (alternatives compared); $*$ = the pairwise comparison will surely trigger inconsistency

Let us now run through all 5 iterations – before running the code – and analyze what should happen:

1. The pairwise comparisons in iteration 1 will trigger inconsistency. Since this is the only stored feedback, it will be immediately removed, and the model construction attempt using an empty history will be executed. This attempt will – obviously – end with a success (100% success rate for *FRS*).
2. The following iteration will add the second pairwise comparison to the history – and it will be the only stored feedback. It is the same preference information as the pairwise comparison in the previous tutorial. Hence, the model should prove compatible, and the sampling success rate should be around 66.66%.
3. The third iteration starts with a history consisting only of feedback no. 2 and feedback no. 3 will be immediately added – causing the inconsistency. The *RemoveOldest* procedure will then start removing stored feedback from the oldest to the newest until compatibility is restored. Unluckily, it's the newest feedback that makes the model incompatible. Therefore, all stored feedback will be removed first. Then, *RemoveOldest* will start adding the removed feedback from the newest to the oldest. Those preference statements that will not violate compatibility will be restored in the preference elicitation history (preserving their original order). Consequently, in the end, the history will consist only of feedback no. 2 (removed but restored later).
4. Iteration no. 4 will introduce a compatible pairwise comparison to the history. It is the same feedback as in the previous tutorial. Hence, a sampling success rate of 33.33% is expected.
5. The last iteration will cause inconsistency. As with iteration no. 3, all stored pairwise comparisons will be removed, and only feedback no. 2 and 4 will be restored in the second phase of the inconsistency handling procedure.

The console output generated by the source code confirms the above predictions – it is up to the reader to delve into the

complete log. For convenience, Figures 34 and 35 present, respectively, the console output focused on the first and last iterations. Both match our expectations. For instance, iteration 0 states that "*inconsistency detected* = true", and the consistency was immediately reintroduced by removing feedback no 1 from the set (100% success rate in constructing). Regarding the last iteration, the log also confirms the predictions. Note that there is an "All states" log part that states "None". *RemoveOldest* can capture reports on all states it visited when reintroducing consistency. State, in this case, means utilizing some particular set of preference statements in the analysis (see the *State* class in *inconsistency* package). These states are not saved by default, but this option can be turned on by setting the proper flag (see the code comments).

```
History (tutorial)
[Pairwise comparison: A0 PREFERENCE A1 ; id = 0 ; iteration = 0 ;
Iteration = 0
Date and time (decision-making context) = [13.11.2024 18:32:37:101
Date and time (report creation) = [13.11.2024 18:32:37:103]
Inconsistency detected = true
The number of generated models = 0
Elapsed time = 154 ms
Success rate in preserving = 0.0
Models preserved between iterations = 0
Models rejected between iterations = 0
Success rate in constructing = 0.0
Accepted newly constructed models = 0
Rejected newly constructed models = 1000000
Normalizations were updated = true

Inconsistency handling (final report):
The number of attempts = 1
All states:
  None
Consistent state:
  State for attempt = 1
  Data on model bundle:
    Iteration = 0
    Date and time (decision-making context) = [13.11.2024 18:32:37
    Date and time (report creation) = [13.11.2024 18:32:37:269]
    Inconsistency detected = false
    The number of generated models = 1000
    Elapsed time = 0 ms
    Success rate in preserving = 0.0
    Models preserved between iterations = 0
    Models rejected between iterations = 0
    Success rate in constructing = 1.0
    Accepted newly constructed models = 1000
    Rejected newly constructed models = 0
    Normalizations were updated = false
  Data on preference information:
    None
```

Figure 34: Expected console output (first iteration)

```
Iteration = 4 ======================
History (tutorial)
[Pairwise comparison: A4 PREFERENCE A5 ; id = 1 ; iteration = 1 ; date time =
[Pairwise comparison: A12 PREFERENCE A13 ; id = 3 ; iteration = 3 ; date time
[Pairwise comparison: A16 PREFERENCE A17 ; id = 4 ; iteration = 4 ; date time
Iteration = 4
Date and time (decision-making context) = [14.11.2024 14:49:36:197]
Date and time (report creation) = [14.11.2024 14:49:36:197]
Inconsistency detected = true
The number of generated models = 0
Elapsed time = 151 ms
Success rate in preserving = 0.0
Models preserved between iterations = 0
Models rejected between iterations = 1000
Success rate in constructing = 0.0
Accepted newly constructed models = 0
Rejected newly constructed models = 1000000
Normalizations were updated = false

Inconsistency handling (final report):
The number of attempts = 6
All states:
  None
Consistent state:
  State for attempt = 6
  Data on model bundle:
    Iteration = 4
    Date and time (decision-making context) = [14.11.2024 14:49:36:197]
    Date and time (report creation) = [14.11.2024 14:49:36:584]
    Inconsistency detected = false
    The number of generated models = 1000
    Elapsed time = 0 ms
    Success rate in preserving = 0.518
    Models preserved between iterations = 518
    Models rejected between iterations = 482
    Success rate in constructing = 0.3333333333333333
    Accepted newly constructed models = 482
    Rejected newly constructed models = 964
    Normalizations were updated = false
  Data on preference information:
    [Pairwise comparison: A4 PREFERENCE A5 ; id = 1 ; iteration = 1 ; date tim
    [Pairwise comparison: A12 PREFERENCE A13 ; id = 3 ; iteration = 3 ; date t
```

Figure 35: Expected console output (first iteration)

### *4.3. Model system [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t3_models_updater_module.t3_complex_example***

This tutorial is a direct continuation of the previous one. It exemplifies how all the steps done in the previous tutorial can be accomplished by employing the *ModelSystem* class. This class must be supplied with the definition of a preference model, preference model constructor, and inconsistency handler upon creation. Then, all the relevant processes can be triggered by calling its "*updateModel()*" method. The source code is straightforward and introduces little new information compared to the previous tutorial. Hence, it is left up to the reader to analyze the code.

## 5. Decision Support System

The *DecisionSupportSystem* class is considered the highest in the hierarchy (see Figure 1; see *system.ds* package). It maintains all the sub-systems, i.e., *PreferenceElicitationModule* and

*ModelsUpdaterModule*, takes control over their execution, and provides a more straightforward means for realizing all designed processes.

Let us start first with constructing an object instance. As usual, it is done via the params container, showcased in Listing 33 for convenience. Notably, its fields introduce nothing new. The user is expected to provide such data as criteria, interaction trigger, refiner, reference sets constructor, feedback provider, and decision makers' bundles. All these fields – except for the last, are used to establish the preference elicitation module. In turn, the *DMBundle* is an auxiliary container-like class that provides all essential data required to establish decision maker systems (the object creation is the prerogative of the decision support system).

Listing 33: DecisionSupportSystem.Params class

```
/**
 * Params container.
 */
public static class Params
{
  /**
   * Represents a consistent family of
       criteria.
   */
  public Criteria _criteria;

  /**
   * Interaction trigger.
   */
  public InteractionTrigger
      _interactionTrigger;

  /**
   * Refiner object.
   */
  public Refiner _refiner;

  /**
   * Reference sets constructor.
   */
  public ReferenceSetsConstructor
      _referenceSetsConstructor;

  /**
   * Object responsible for constructing a
       feedback given the reference sets.
   */
  public FeedbackProvider _feedbackProvider
      ;

  /**
   * Decision maker's data bundles.
   */
  public DMBundle[] _dmBundles;

  [...]
}
```

Listing 34 provides the most essential code snippet from *DMBundle* class. The showcased fields are self-explanatory. The "*_name*" must be employed to associate a decision maker's string identifier (provided via the constructor). **Important note**: the provided identifiers must be, of course, unique and consistent with other system's components (e.g., reference set constructors, which may be appointed to a specific decision maker via the string identifier). The system will anyway check the validity of the data upon initialization. The latter field – "*_modelBundles*" – specifies model-bundles, where each bundle provides the data required to instantiate the decision maker's model system.

Listing 34: DMBundle class

```
/**
 * Bundle object used for wrapping all
     essential data required to establish a
     single decision maker system
 * (see {@link system.dm.
     DecisionMakerSystem}).
 *
 * @author MTomczyk
 */
public class DMBundle
{
  /**
   * Decision maker's name.
   */
  public final String _name;

  /**
   * Model bundles.
   */
  public ModelBundle<? extends
      AbstractInternalModel>[] _modelBundles
      ;

  [...]
}
```

The model-bundle class (most relevant code snippets) is provided in Listing 35. As expected, its three central fields allow setting the preference model definition, model instances constructor, and inconsistency handler.

Listing 35: ModelBundle class

```
**
 * Bundle object used for wrapping all
     essential data required to establish a
     single model system
 * (see {@link system.dm.
     DecisionMakerSystem}).
 *
 * @author MTomczyk
 */
public class ModelBundle <T extends
    AbstractInternalModel>
{
  /**
   * Preference model used to represent the
       decision maker's preferences.
   */
```

```
  public IPreferenceModel<T>
      _preferenceModel;

  /**
   * Preference model constructor.
   */
  public IConstructor<T> _modelConstructor;

  /**
   * Object responsible for handling
       inconsistency that can occur when
       building the model(s) being in line
       with the
   * decision maker's aspirations.
   */
  public IInconsistencyHandler<T>
      _inconsistencyHandler;

  [...]
}
```

Notably, the *DecisionSupportSystem* requires some parameterization. It is recommended to the reader, thus, to introduce their own shortcuts that will allow for hiding the parameterization. It is common in hierarchical structures. For instance, the initial release of JECDM provides *DSSParamsProvider* class (*system.ds* package), which provides (currently only one) method that aids in quickly establishing system's params container dedicated to (i) one decision maker with (ii) one preference model and (iii) one decision-maker-based feedback provider. The method is brought in Listing 36 for convenience.

Listing 36: DSSParamsProvider class

```
/**
 * Constructors a DSS params container that
     involves one decision maker with one
    preference model, one DM-based feedback
     provider,
 * and one interaction rule.
 *
 * @param criteria
    consistent family of criteria
 * @param DM                        decision
    maker's identifier
 * @param interactionRule
    interaction rule
 * @param referenceSetConstructor reference
     set constructor
 * @param dmFeedbackProvider       DM-based
    feedback provider
 * @param preferenceModel
    preference model used
 * @param modelConstructor         model
    instance constructor
 * @param <T>                      internal
    preference model definition
 * @return parameterized {@link
    DecisionSupportSystem.Params} object
 */
public static <T extends
    AbstractInternalModel>
    DecisionSupportSystem.Params
    getForSingleDecisionMakerSingleModel
    ArtificialProvider(
  Criteria criteria,
  String DM,
  IRule interactionRule,
  IReferenceSetConstructor
      referenceSetConstructor,
  IDMFeedbackProvider dmFeedbackProvider,
  IPreferenceModel<T> preferenceModel,
  IConstructor<T> modelConstructor)
{
  DecisionSupportSystem.Params p = new
      DecisionSupportSystem.Params();
  p._criteria = criteria;
  p._interactionTrigger = new
      InteractionTrigger(interactionRule);
  p._refiner = Refiner.getDefault();
  ReferenceSetsConstructor.Params pRSC =
      new ReferenceSetsConstructor.Params();
  pRSC._commonConstructors = new LinkedList
      <>();
  pRSC._commonConstructors.add(
      referenceSetConstructor);
  p._referenceSetsConstructor = new
      ReferenceSetsConstructor(pRSC);
  p._feedbackProvider = FeedbackProvider.
      getForSingleDM(DM, dmFeedbackProvider)
      ;
  DMBundle dmBundle = new DMBundle(DM);
  dmBundle._modelBundles = new ModelBundle
      <?>[1];
  ModelBundle<T> modelBundle = new
      ModelBundle<>();
  modelBundle._preferenceModel =
      preferenceModel;
  modelBundle._modelConstructor =
      modelConstructor;
  modelBundle._inconsistencyHandler = new
      RemoveOldest<>();
  dmBundle._modelBundles[0] = modelBundle;
  p._dmBundles = new DMBundle[]{dmBundle};
  return p;
}
```

### 5.1. *Example use case #1 [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t4_decision_support_system.t1_use_case***

This tutorial exemplifies how to create and use a simple decision support system. The code is mainly about the system's parameterization, so it does not introduce much new. Thus, it is left to the reader to get acquainted with the code. However, Listing 37 brings – for convenience – the code snippet showcasing how to use the system. As expected, using *DecisionSupportSystem* is relatively straightforward once it is instantiated. First, it is essential to call the "*notifySystemStarts()*" method (if

not called, the system will throw an informative error when using its central functionalities). Next, running the whole preference elicitation process (*PreferenceElicitationModule*) and conditionally preference learning (only if the interactions were triggered and successfully conducted; *ModelsUpdaterModule*) can be done just by calling the "*executeProcess(decision-making context params container)*" method. As with other system components, the method returns a report on the execution of the underlying processes. It is recommended to run the source code and analyze the printed log.

Listing 37: Use case #1 (using DecisionSupportSystem)

```
// Create DSS
DecisionSupportSystem DSS;
try
{
  DSS = new DecisionSupportSystem(pDSS);
  // Important note: use must call ''
      notifySystemStarts'' before using DSS'
      s functionalities
  DSS.notifySystemStarts();

    // Example alternatives (performance
        vectors):
  double[][] evaluations = new double
      [][]{{1.0d, 0.0d}, {0.75d, 0.25d},
      {0.5d, 0.5d}, {0.25d, 0.75d}, {0.0d,
      1.0d}};

  // Create the context:
  DMContext.Params pDMC = new DMContext.
      Params();
  pDMC._currentIteration = 0;
  pDMC._currentOS = new ObjectiveSpace(new
      Range[]{Range.getNormalRange(), Range.
      getNormalRange()}, new boolean[2]);
  pDMC._osChanged = false;
  pDMC._currentAlternativesSuperset = new
      Alternatives(Alternative.
      getAlternativeArray("A", evaluations))
      ;
  pDMC._R = R;

  // Central function: executes the
      preference elicitation step and (if
      interactions were triggered) models
      updater module.
  Report report = DSS.executeProcess(pDMC);

  // Print the report:
  report.printStringRepresentation();

} catch (DecisionSupportSystemException e)
{
    throw new RuntimeException(e);
}
```

### 5.2. *Example use case #2 [T]*

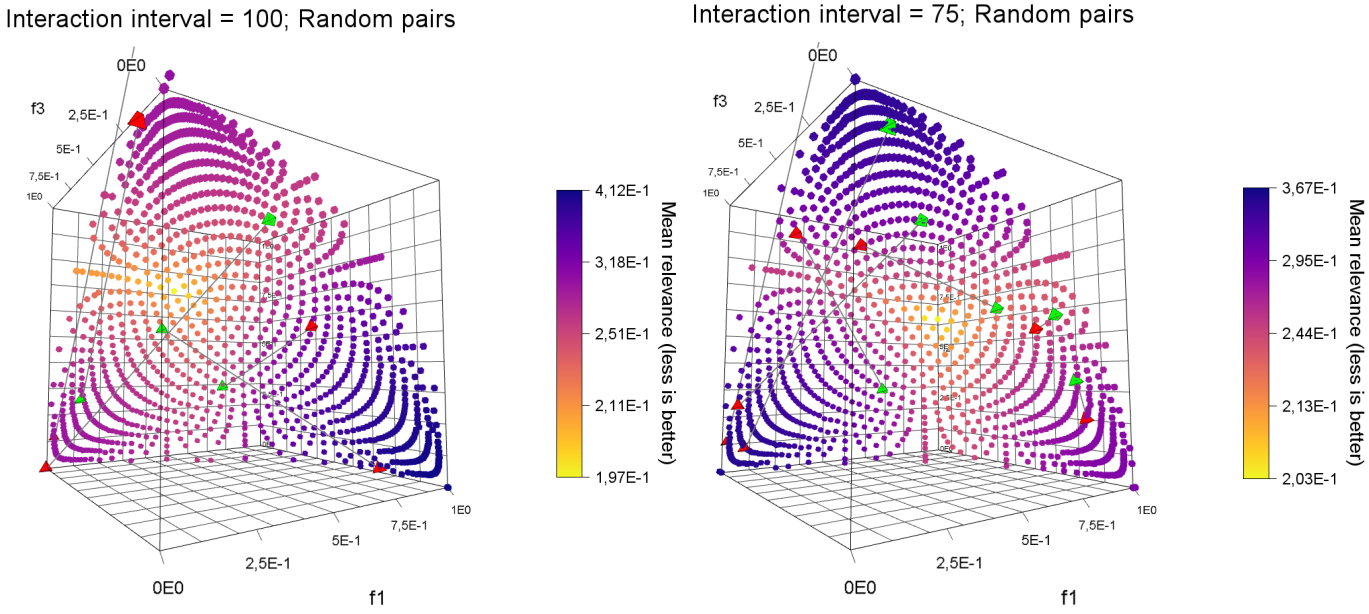
Source package: *t1_t10.t4_decision_support_module.* ***t4_decision_support_system.t2_use_case***

This complex tutorial example is similar to the one presented in Section 3.5. Specifically, the code runs the MOEA/D algorithm. This optimizer is implicitly coupled with *DecisionSupportSystem* – implicitly means that the latter is not formally an integral part of MOEA/D. After each generation, a decision-making context is established. It considers the current population the main input alternatives superset for *DecisionSupportSystem*. The system will then execute the preference elicitation step and, if triggered, will update the decision maker's internal models ($L_\infty^{\boldsymbol{w}}$-norms). MOEA/D will not receive feedback from *DecisionSupportSystem* – so it will proceed as usual. However, the feedback will be used to colorize depicted population members (the visualization is run). For this purpose, the gradient will be used, and the color will depend on the average score attained by an alternative given all constructed preference model instances. Thus, the color can highlight alternatives with a more significant comprehensive relevance score. Additionally, as in the example in Section 3.5, all pairwise comparisons will be depicted. The general scenario setting is as follows:

- MOEA/D is run for 500 generations and the population size of 1326.
- A single decision maker is considered. their preferences are modeled using $L_\infty^{\boldsymbol{w}^{DM}}$-norms.
- The interactions are conducted at regular intervals (75 by default = 7 interactions).
- The system selects a random pair of alternatives created by MOEA/D for comparison.
- The artificial decision maker is modeled using $L_\infty^{\boldsymbol{w}^{DM}=1/3,1/3,1/3}$-norm.
- The *FRS* method is used to construct internal models (sampling limit = 1000000; models to sample = 1000).
- The 3D visualization is run to illustrate populations constructed by MOEA/D. Each specimen is colored using a gradient. The color is determined based on the specimen's average relevance given the constructed internal models (nonlinear scale is used; gamma correction is applied to increase contrast around more relevant alternatives). Additionally, collected pairwise comparisons are depicted using lines ended with markers (green marker = alternative is preferred; red = not).

Note that the scenario setting requires creatively customizing the display ranges and data updater. Therefore, it is recommended that the reader examine the relevant code parts.

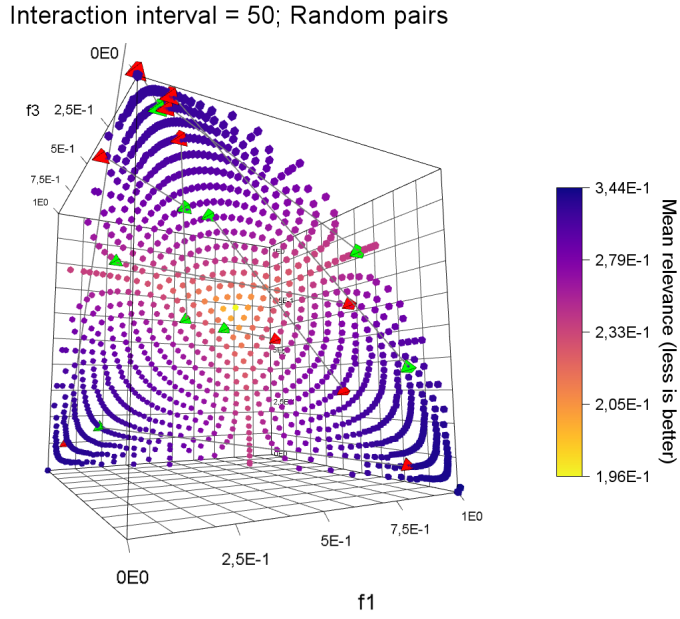
Figure 36 depicts the final visualization outcome and the results attained with slightly modified scenario settings. Specifically, Figure 36a showcases the result when the interaction interval was set to 100 → 5 interactions, Figure 36b concerns



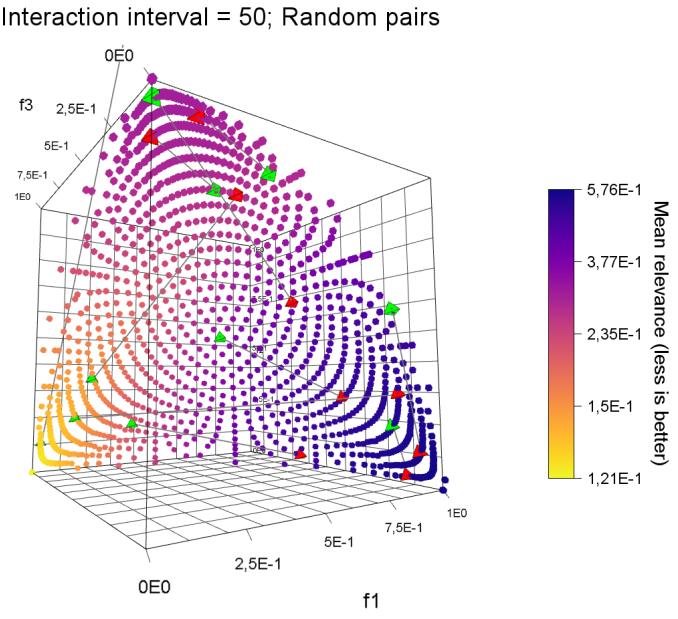

(a) 5 pairwise comparisons; $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$




(b) 7 pairwise comparisons; $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$




(c) 10 pairwise comparisons; $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$




(d) 10 pairwise comparisons; $\boldsymbol{w}^{DM} = [0.5, 0.3, 0.2]$

Figure 36: Expected final visualization

the default setting, while Figure 36c provides outcomes for the most intense setup: 10 interactions (interaction interval of 50). Additionally, Figure 36d concerns the default scenario, but with the weights for the artificial decision maker changed to $[0.5, 0.3, 0.2]$.

As explained, the colors used to render the population can be understood as alternatives' relevance scores. In this example, bright yellow colors are preferred over the blue ones. Notably, for all four cases, the indications yielded by the coloring process are consistent with the expectations regarding which alternatives should be the most preferred by the decision maker. For the decision maker modeled using the weight vector of $[1/3, 1/3, 1/3]$, we could expect that the centrally located alternatives will be highlighted as the most relevant ones. In turn, alternatives that minimize primarily $f_1$ are recommended when the vector is set to $[0.5, 0.3, 0.2]$ (Figure 36d). Lastly, it is evident that the more feedback is provided, the better the method can understand the true decision maker's preferences. For settings with more frequent interactions, the specimens shaded using the yellow color occupy a more concentrated area in the Pareto front, pointing more precisely to the most relevant alternatives. In turn, for the setting with just 5 interactions, it was not enough to accurately understand preferences.

## 6. Implemented hybrid algorithms

A few of the preceding tutorial sections employed the decision-support module and evolutionary computation altogether, but they assumed that the interaction between these components is one-way. i.e., the former was fed with alternatives generated by the optimizer, but the optimized made no use of the learned preferences. This section, however, showcases a few selected, well-acknowledged, and implemented in JECDM, preference-based interactive evolutionary algorithms for multi-objective optimization. Such optimizers assume a two-way cooperation between the involved components. Precisely, after the preferences are learned, the methods suitably direct the search process to **converge the evolved population towards solutions highly preferred by the decision maker, which is the ultimate goal of a decision-reaching process executed in an optimization environment**.

As for the methods implemented for the initial release of JECDM, these are: IEMO/D [8], NEMO-0 and NEMO-II (their variants based on $L_\alpha^w$-norms, [6, 8]), and CDEMO and DCEMO [10]. **Note that** the reader is referred to the respective papers for more information about these methods – the following sections will briefly cover only the basics. Moreover, it is assumed that the reader already gets acquainted with the preceding tutorial documents and, thus, can inspect the codes on their own to understand how these approaches are implemented JECDM – no extensive diagrams or pieces of code will be brought here. Before proceeding, however, it is worth listing a few classes that are common to all these implementations and may be helpful when designing new hybrid algorithms:

- *AbstractEMOInteractiveBundle* (see *emo.interactive* bundle): This bundle-like class extends *AbstractEMOBudle* and provides additional parameterization layer for the interactive methods. It is responsible for (i) maintaining the *DecisionSupportSystem* object (coupled with the developed algorithm), maintaining the *AbstractDMCParamsConstructor* object (auxiliary class aiding in creating the decision-making context; see *emo.interactive.utils.dmcontext* package; see the point below), and instantiating the *InitStartsRunDSS* (see phases package) as the default "init starts" phase (see the point below).

- *AbstractDMCParamsConstructor*: As already explained, the instantiable extensions of this class are responsible for creating the decision-making context (see *DefaultConstructor* class). Important note: these classes implement the already known *IOSChangeListener* interface. Upon notification, the object sets the "OS changed" flag to true, used when creating the context to inform the *DecisionSupportSystem* that it must recalculate the normalization functions. When designing own methods, it is recommended to bind this listener with the "update objective space" phase.

- The default *InitStartsRunDSS* phase is an initial phase (run before everything else), and it is responsible for calling the "*notifySystemStarts()*" method of the decision support system.

- *AbstractInteractAndPrepareStep*: This is a default implementation of the "prepare step" for interactive methods. When coupling the evolutionary process with the decision support system, the relevant question is: When should the former communicate with the latter (note that the latter is not operating independently; it has to be called by the former)? In the developed methods, the communication is done exactly during the "prepare step" phase (**only when the current steady state repeat number is 0!**). It is a reasonable decision because all the relevant processes can be thus executed before conducting an actual step of the evolutionary algorithm (select parents, reproduce, assess specimens, etc.). **Important note:** It also means that, in the supplied implementations, there is no way of triggering interactions in generation 0 (the beginning of generation 1 is the first time moment when the interaction can be triggered; which can be reflected in suitably parameterizing the interaction rules, e.g., *IterationInterval* with the starting iteration of 1). It also makes sense because, practically, there is no reason to interact if you do not have any specimens ("init begins" phase). In turn, the "init ends" of generation 0 (the initial population is already constructed) effectively can be treated as the "prepare step" for generation 1. **Additional note**: *AbstractInteractAndPrepareStep* implements the *IOSChangeListener*. It is supposed to be delivered along with *AbstractDMCParamsConstructor* (but after it) as one of the method's listeners. If the listener's action is triggered, the default implementation executes the primary process of the *ModelsUpdaterModule*. The rationale is that the current models become invalid if the known relevant objective space bounds are changed (normalization functions should be revised, thus internal models). In such a case, the default constructor – *FRS* – first updates the model's normalization functions (when registering the revised decision-making context). Then, it checks whether they remain compatible. Intuitively, if the normalization function changes are small, the model's chances to remain compatible are significant. This way, the computational burden imposed by constantly changing known objective space bounds can be significantly alleviated (the FRS will not have to do much). **Last note**: after the models are updated, the listener delegates processing to its *protected void doInternalUpdate(Report report, LinkedList<DM> dm, EA ea) throws PhaseException* method. A concrete algorithm can implement this method to react to the changes in the preferences learned.

The following sections showcase how to instantiate particular methods and use them. All the employed experimental settings are the same for all methods, and the codes that establish them are relatively self-explanatory. Nonetheless, some of the settings are listed below for convenience:

- The algorithms are run for 500 generations (from 0 to 499).

- Two scenarios regarding the number of objectives involved are considered: $M = 2$ and 3 objectives.

- A simple DTLZ2 test was employed but with a slightly greater-then-typical-number of 50 distance-related variables to slightly slow down the convergence pace.
- The interactions are triggered after each 50 generations, starting from generation number 50 (thus, are interactions triggered in generation no. 50, 100,...450).
- The artificial decision maker is employed to simulate feedback on pairwise comparison questions. Its value system is represented using $L_\infty^{\boldsymbol{w}^{DM}}$-norm with $\boldsymbol{w}^{DM} = [0.5, 0.5]$ (case #1 for $M = 2$), $\boldsymbol{w}^{DM} = [0.25, 0.75]$ (case #2 for $M = 2$), $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$ (case #1 for $M = 3$), and $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$ (case #2 for $M = 3$).
- The methods will use $L_\infty^{\boldsymbol{w}}$-norm to represent the decision maker's preferences (the model aligns with the decision-making policy of the artificial decision maker, hence, inconsistencies are not expected).

### 6.1. IEMO/D [T]

Source package: *t1_t10.t4_decision_support_module.* ***t5_hybrid_algorithms.t1_iemod***

IEMO/D is a decomposition-based interactive method. It was primarily designed to work with a single decision maker, $L_\alpha^{\boldsymbol{w}}$-norms (used as a preference model), and pairwise comparisons (used as feedback). However, a generalized implementation of this algorithm is available in this framework in *emo.interactive.iemod* package. Specifically, any instantiated decision support system can be coupled with IEMO/D as long as it is set to operate with one decision maker and is capable of decomposing their preferences and representing them as a series of scalarizing functions (extensions of *AbstractValueInternalModel*). The latter requirement is fundamental for IEMO/D. This method makes use of the fact that "decomposition" in evolutionary multi-objective optimization plays a similar role to "disaggregation" in preference learning – to use a set of functions to represent various possibilities (in diversifying the evolutionary search or representing the decision maker's aspirations). In a nutshell, IEMO/D explicitly treats the scalarizing functions generated during the preference learning step as optimization goals used in the decomposition-based search (inherently, IEMO/D is an extension of MOEA/D). Naturally, these goals are updated each time new feedback is provided and the internal models are constructed (this general idea is presented in Figure 37). In this view, IEMO/D elegantly connects the two worlds: preference learning and decomposition-based optimization. Additionally, IEMO/D implements robustness analysis in this way, increasing the credibility of derived recommendations. Specifically, instead of focusing the search on a single point, it aims at approximating the part of the Pareto front that consists of solutions being potentially the most relevant to the decision maker [5].

Readers interested in understanding how IEMO/D was implemented in JECDM are referred to the relevant codes in the *emo.interactive.iemod* package. Note that the *IEMODBundle* employs the MOEA/D's components to a large extent. Also, inspecting the "*doInternalUpdate*" method" in *IEMODInteractAndPrepareStep* phase is recommended. It replaces the optimization goals maintained in *MOEADGoalsManager* each time the internal models are updated. Such replacement also requires re-establishing the neighborhood and re-assigning specimens to goals. Finally, the instance of IEMO/D can be obtained via the auxiliary *IEMOD* class (extends *EA*; available in *emo.interactive.iemod* package).

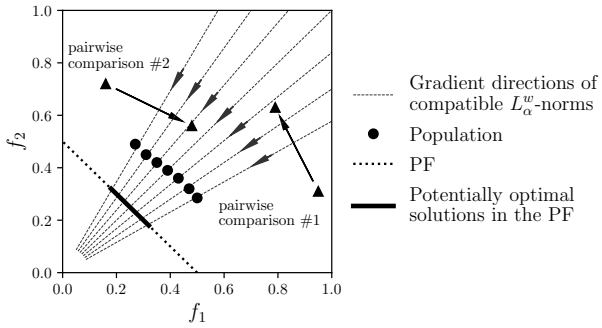
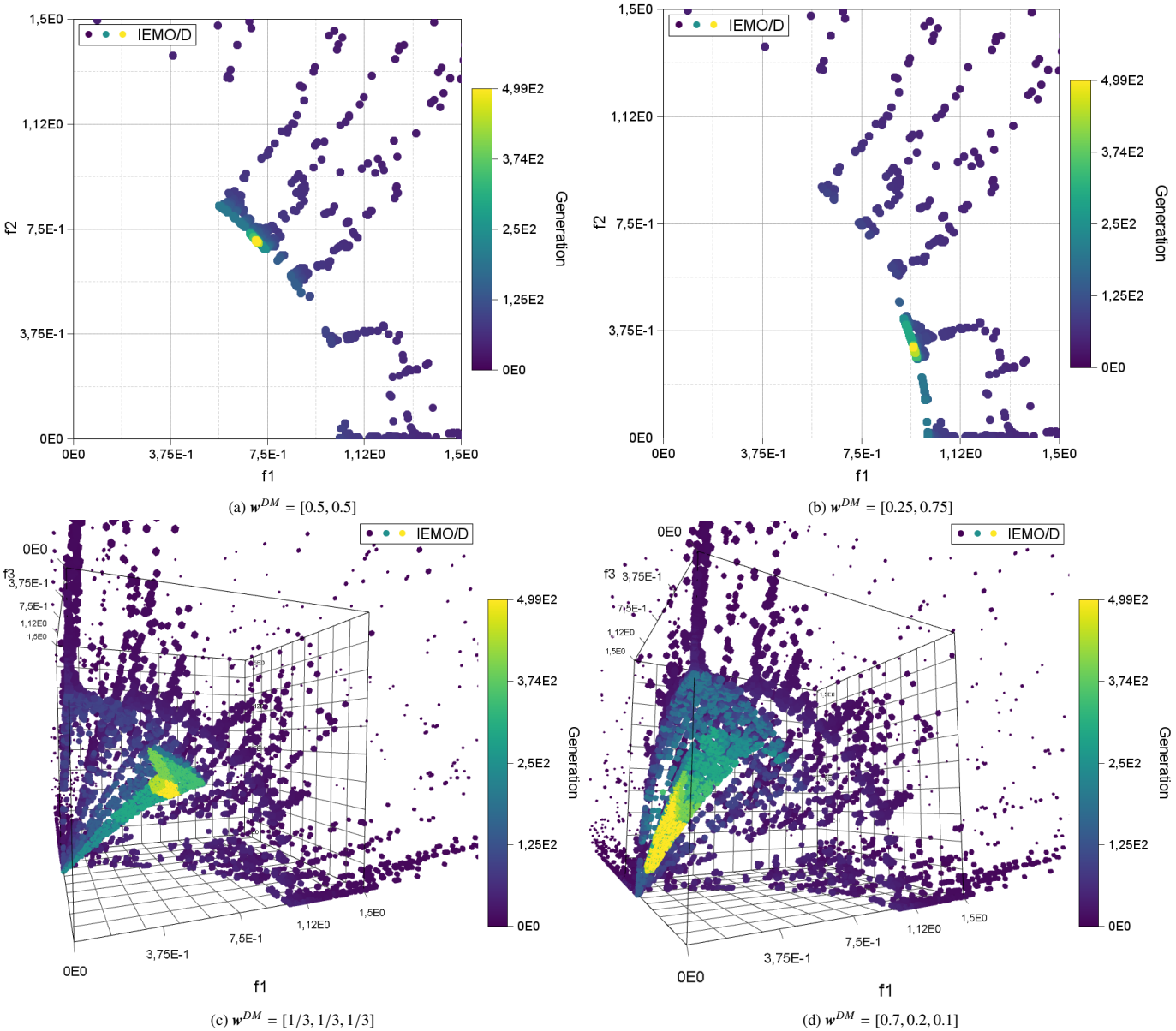



Figure 37: IEMO/D – the general idea

The available source codes are **Tutorial1a** (for $M = 2$) and **Tutorial1b** (for $M = 3$). Note that the default policy for selecting pairs to be compared is **RandomPairs**. However, **Tutorial1a** shows how to employ a more advanced technique based on calculating Pairwise Winning Indices (PWI; see the lines of code that are commented out). This procedure must be coupled with a preference model to represent the decision maker's preferences (assuming it will be supplied with multiple internal models during the preference learning phase). It selects a pair to be presented to the decision maker that maximizes the potential information gain that can be received from their feedback (see [3] for more information on this procedure). Note that, however, this procedure imposes a much greater computational burden than *RandomPairs*.

The expected outcomes for all 4 scenarios considered (2 configurations of the number of objectives × 2 configurations of $\boldsymbol{w}^{DM}$) are illustrated in Figure 38. The plots reveal all populations constructed throughout all generations using a gradient. Notably, IEMO/D was fully capable of converging towards some more preferred areas in the Pareto-front (this required suitably learning the decision maker's preferences and accordingly adjusting the search). Observe that the populations constructed throughout subsequent generations often form characteristic lines (clearly visible in Figure 38a and 38b). It was expected. The populations followed the most promising gradient descent directions of the employed Chebyshev functions used as optimization directions (these directions constitute lines). Further, observe that the populations formed consolidated areas that were shrinking during subsequent interactions. It was due to implementing the robustness concern, which, in IEMO/D, results in approximating the part of objective space containing potentially the most promising solutions for the decision maker at each stage of the preference learning process.




(a) $\boldsymbol{w}^{DM} = [0.5, 0.5]$

(b) $\boldsymbol{w}^{DM} = [0.25, 0.75]$

(c) $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$

(d) $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$

Figure 38: Expected final visualization (IEMO/D)

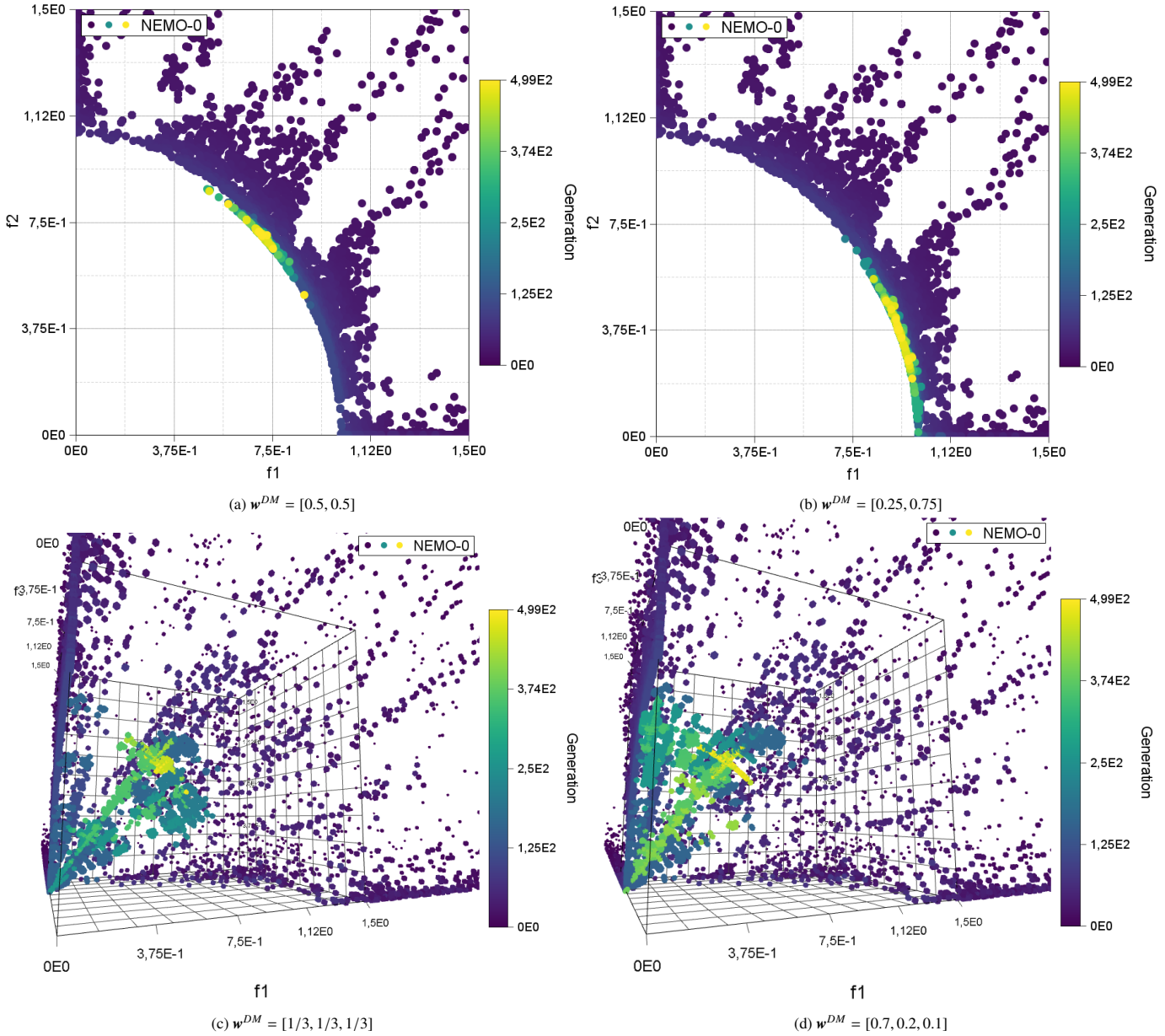
### 6.2. *NEMO-0 and NEMO-II [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t5_hybrid_algorithms.t2_nemo0_nemoii***

NEMO-0 and NEMO-II are interactive methods founded on NSGA-II algorithm. They suitably revise NSGA-II's sorting criteria to steer the evolution in line with the decision maker's aspirations. These algorithms' implementations are available in the *emo.interactive.nemo* package (they mainly introduce sorting phases that are extensions of *NSGAIISort*). Note that, however, the original implementations proposed in [6] assumed using the utility value function model (as the preference model) and linear programming (as a means of exploiting it). The JECDM's implementation, however, allows the use of any value-based model as long as it can be decomposed into a series of compatible model instances (linear programming-based implementations are in plans for future developments). By default, these implementations can be coupled with $L^{\boldsymbol{w}}_{\alpha}$-norms and *FRS*.

Regarding the ways both methods impose the search in line with the decision maker's preferences, they are as follows:

- NEMO-0: This method revises the secondary sorting criterion of NSGA-II. Instead of employing the crowding-distance indicator to promote well-spread solutions (and to approximate the whole Pareto front), NEMO-0 derives the so-called most representative preference model instance to rank them according to their relevance to the decision maker. In short, instead of using all derived compatible




(a) $\boldsymbol{w}^{DM} = [0.5, 0.5]$

(b) $\boldsymbol{w}^{DM} = [0.25, 0.75]$

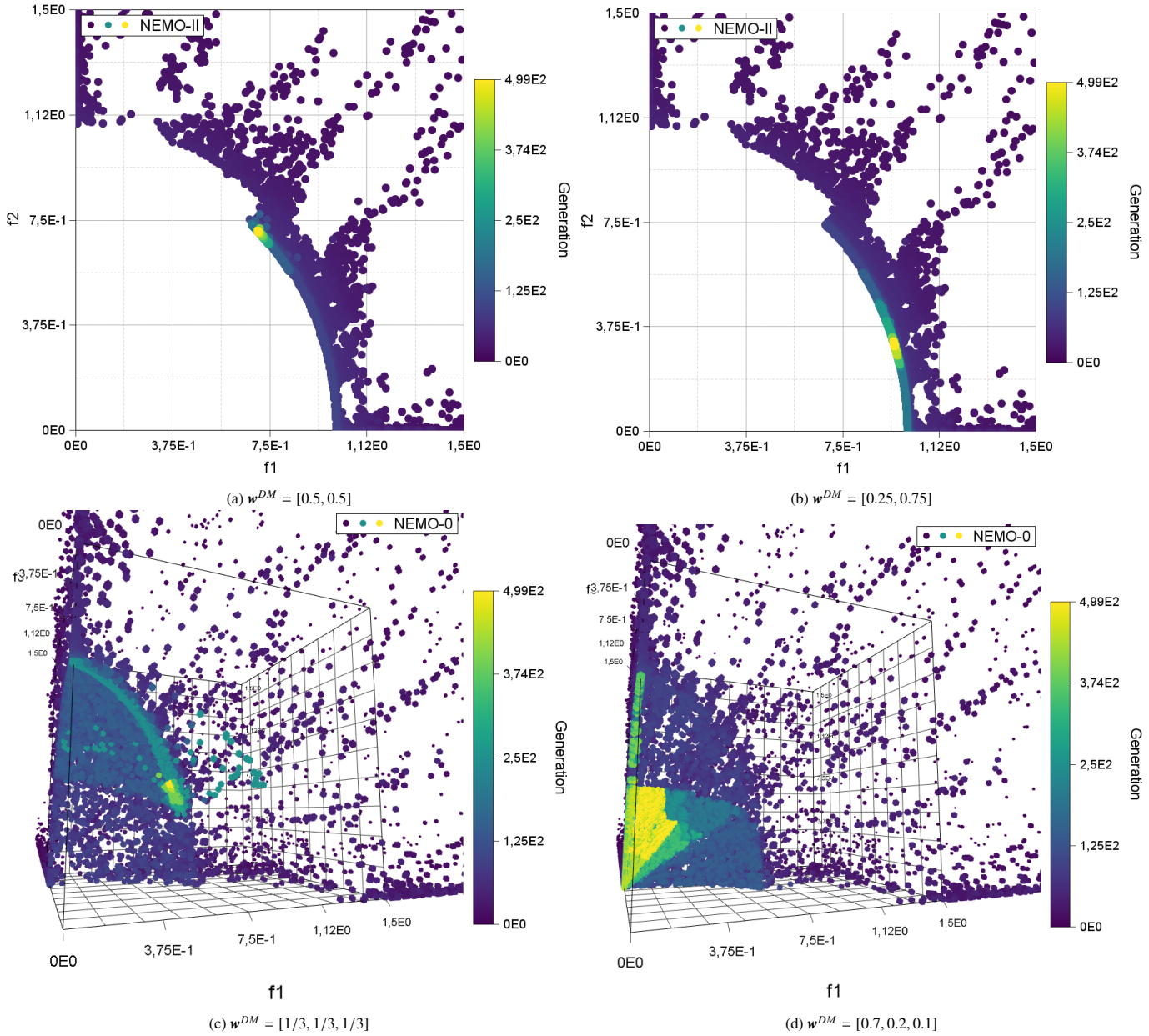
(c) $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$

(d) $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$

Figure 39: Expected final visualization (NEMO-0)

model instances – just one, i.e., the representative, is utilized (see the default constructor *RepresentativeModel* in *model.constructor.value.frs.representative* package). Unfortunately, such a search does not follow the robustness paradigm (thus, the final recommendations may be less credible). Additionally, the representative function may change to a large extent after each preference learning step (the changes are less extreme in later stages of the decision-reaching process when the decision maker's aspirations are well-understood). This means that the overall evolutionary search imposed in this way may be less "stable" as the population will be forced to frequently adapt to different fitness functions.

- NEMO-II: The updated NEMO method resolves the abovementioned issues (not credible recommendations, not stable evolutionary search). It is done by employing the relation of potential optimality when determining the most promising solutions. This relation is used as a replacement for the dominance relation used in the primary non-dominated sorting (the relation of potential optimality enriches the dominant relation; convergence towards relevant solutions will be imposed already at this step). As for the secondary sorting criterion, the original crowding distance indicator is still used. Intuitively, NEMO-II will aim to well-approximate part of the Pareto front that potentially contains the most relevant solutions to the decision maker. Thus, NEMO-II aims to achieve similar goals to IEMO/D. However, the latter method is implemented




(a) $\boldsymbol{w}^{DM} = [0.5, 0.5]$

(b) $\boldsymbol{w}^{DM} = [0.25, 0.75]$

(c) $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$

(d) $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$

Figure 40: Expected final visualization (NEMO-II)

within a much better evolutionary framework, ensuring thus more efficient optimization.

Both algorithms have dedicated classes *NEMO0* and *NEMOII* that are utilized in this tutorial. The expected final visualizations for both methods are presented in Figure 39 (NEMO-0) and 40 (NEMO-II). They are consistent with the general findings on NEMO-0 and NEMO-II. First, NEMO-0 conducts a less stable search due to trying to converge the whole population towards a single point (which may significantly change after each interaction call). In turn, the outcomes of NEMO-II are similar to those of IEMO/D.

### *6.3. CDEMO and DCEMO [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t5_hybrid_algorithms.t3_cdemo_dcemo***

CDEMO and DCEMO methods are cone-contraction-based methods founded on NSGA-II (their implementations can be found in *emo.interactive.ktscone* package). Both replace the crowding-distance indicator with a function that quantifies with how many pairwise comparisons an inspected solution is compatible. It is assumed that the underlying preference model is a Chebyshev function. Thus – as explained in Section 4.1 – the geometric interpretation is that the pairwise comparisons form so-called preference cones. The more cones contain such

a solution, the more this solution is considered relevant (the reader is referred to [10] for more details). The only difference between CDEMO and DCEMO is in how they prioritize the sorting criteria. As for CDEMO, it first partitions solutions in terms of with how many pairwise comparisons they are compatible. Then, in the case of ambiguity in passing specimens to the next generations, the non-dominated specimens are favored (non-dominated sorting is applied). As for DCEMO, the criteria are inverse. Thus, DCEMO resembles more NSGA-II than CDEMO. Note that neither method employs any incompatibility test.

Both algorithms have dedicated classes *CDEMO* and *DCEMO* that are utilized in this tutorial. Figure 41 (CDEMO) and 42 present expected final visualizations for both methods (DCEMO). Both prove capable of suitably narrowing the search, as imposed by the decision maker's feedback.

### *6.4. Comparative analysis [T]*

Source package: *t1_t10.t4_decision_support_module.* ***t5_hybrid_algorithms.t4_comparatve***

This tutorial performs a simple analysis of all the introduced interactive methods: IEMO/D, NEMO-0, NEMO-II, CDEMO, and DCEMO. **Important note**: As in the previous comparative analyses, it intends to showcase how to use these methods simultaneously, not how to perform a methodologically correct and complete study. This will be covered by the following tutorial document: "*Tutorial 5: Experimentation Module.*"

Before proceeding, it is worth highlighting that the goal of preference-based methods differs from that of *a posterior* ones. Thus, the use of different performance indicators may be required. Given the decision maker's aspirations, such indicators should be centrally focused on quantifying how relevant solutions the optimizer can construct. Such assessment is often done in the literature based on using a model of an artificial decision maker that plays the role of an oracle capable of providing accurate relevance scores. The initial implementation of JECDM provides one such indicator (implementation of the already known *IPerformanceIndicator*): *ValueModelQuality* (see Listing 38 that brings the most relevant code snippets; the class can be found in *indicator.emo.preferencebased* package, in the *EvolutionaryComputation* module). The class is built on *IPreferenceModel*. Specifically, when the *"evaluate"* method is called, each population member is assessed given the preference model. Finally, a statistic summarizing these assessments is returned (*IStatistics* object is used). In this view, *ValueModelQuality* can be customized to a large extent.

Listing 38: ValueModelQuality class)

```
/**
 * This indicator uses a model of a
    preference model ({@link model.
    IPreferenceModel}) to assess population
     members.
 * Then, it returns as statistics
    summarizing the derived relevance
    scores.
 *
 * @author MTomczyk
 */
public class ValueModelQuality<T extends
    AbstractValueInternalModel> extends
    AbstractPerformanceIndicator implements
    IPerformanceIndicator
{
  /**
   * Preference model used to assess
      solutions.
   */
  private final IPreferenceModel<T> _model;

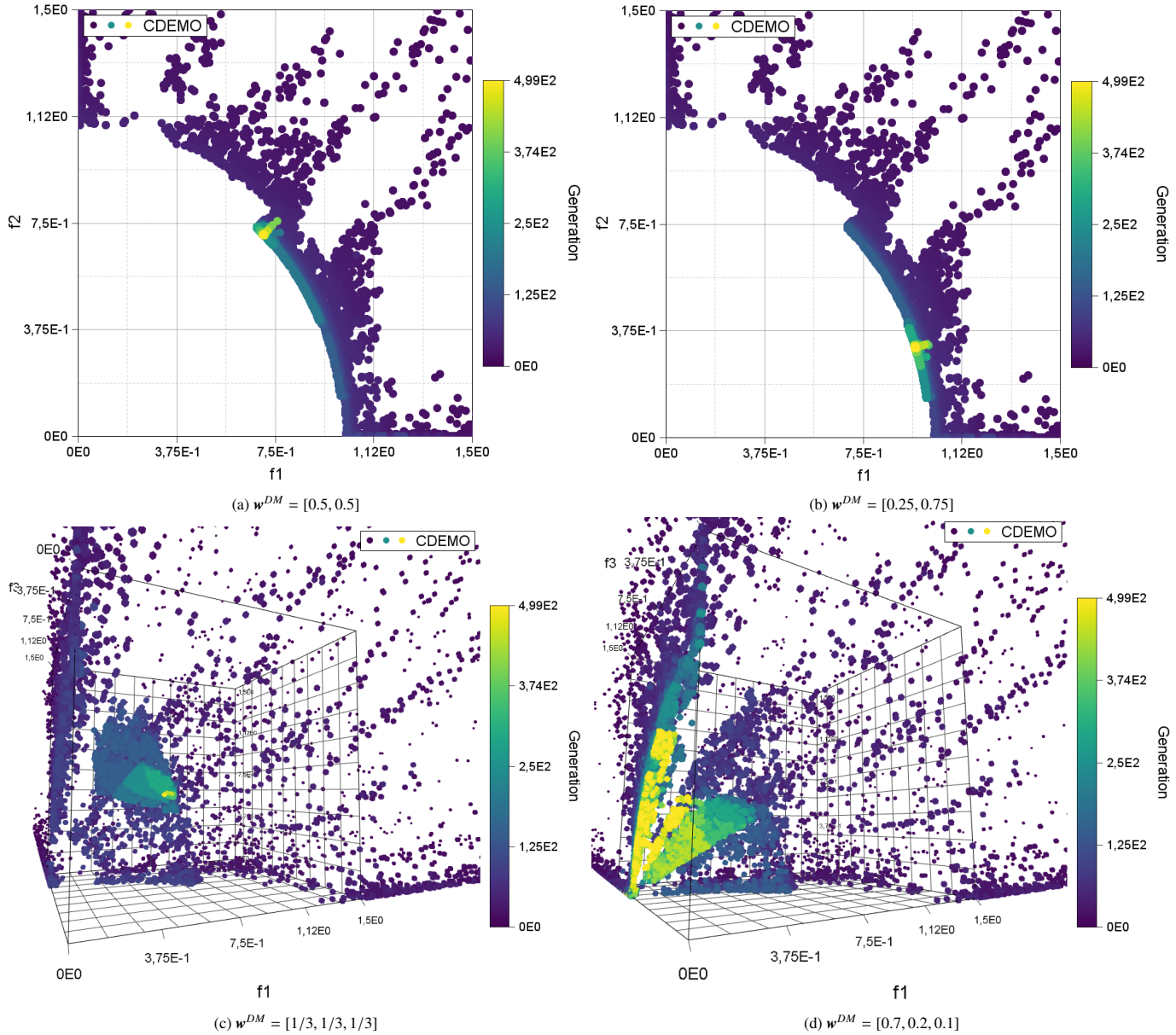
  /**
   * Statistic function used to summarize
      the scores.
   */
  private final IStatistic _statistic;

  [...]

  /**
   * Auxiliary method for assessing the
      performance of EA based on its
      population.
   * The method is to be overwritten.
   *
   * @param population input population
   * @return performance value (0, if an
      exception is thrown by the model)
   */
  protected double evaluate(ArrayList<
      Specimen> population)
  {
    double[] scores = new double[population
        .size()];
    for (int i = 0; i < scores.length; i++)
    {
      try
      {
        scores[i] = _model.evaluate(
            population.get(i).getAlternative
            ());
      } catch (PreferenceModelException e)
      {
        return 0.0d;
      }
    }
    return _statistic.calculate(scores);
  }

  [...]

  /**
   * Method for identifying preference
      direction.
   *
   * @return true - less is preferred
   */
  @Override
  public boolean isLessPreferred()
```




(a) $\boldsymbol{w}^{DM} = [0.5, 0.5]$

(b) $\boldsymbol{w}^{DM} = [0.25, 0.75]$

(c) $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$

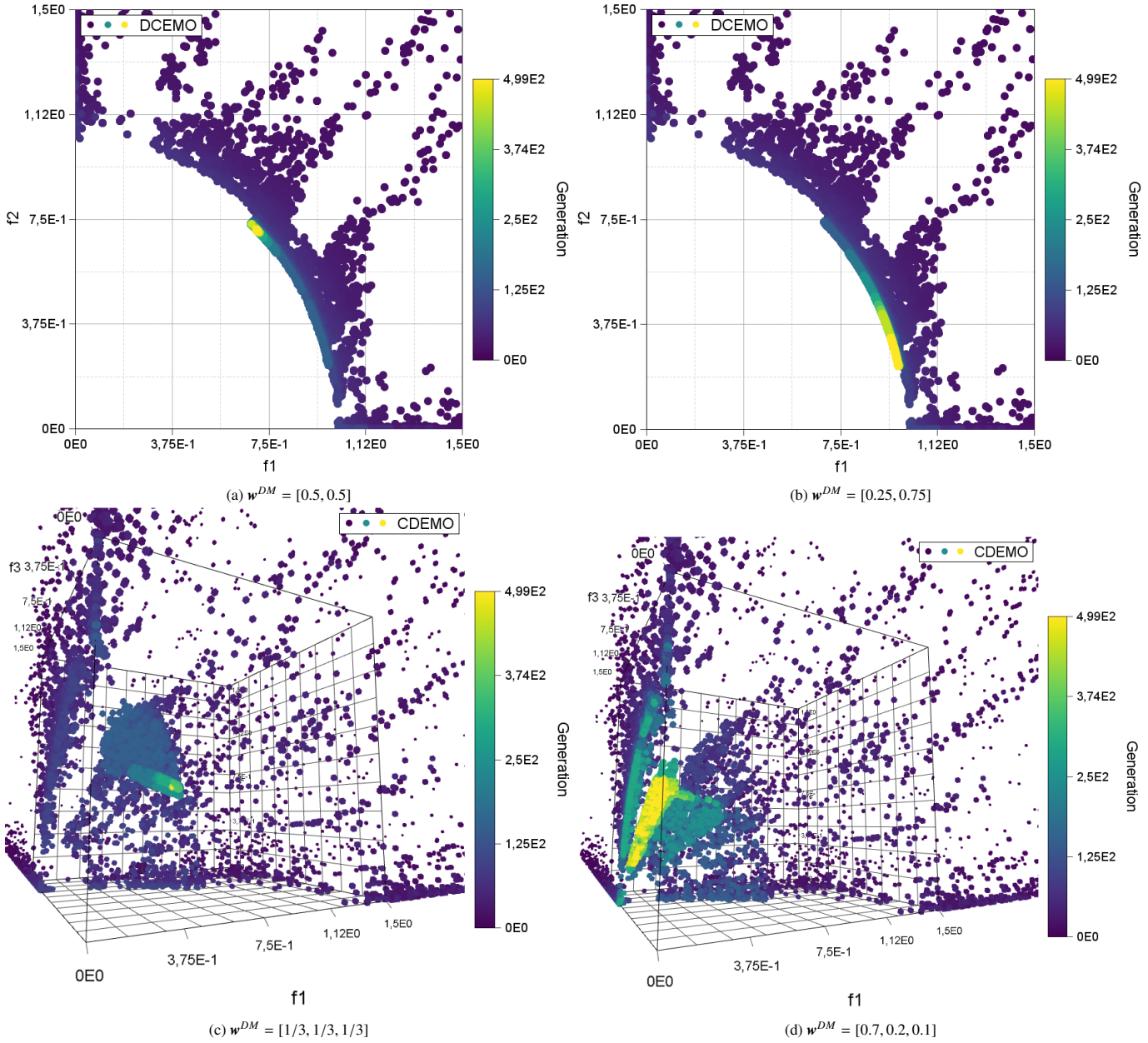
(d) $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$

Figure 41: Expected final visualization (CDEMO)

```
    {
      return _model.isLessPreferred();
    }
}
```

The tutorial's source codes (**Tutorial4a-d**) concern the following (simple) experimental setup:

- The number of objectives $M$ equals 2 (**Tutorial4a**), 3 (**Tutorial4ab**), 4 (**Tutorial4c**), 5 (**Tutorial4d**).
- The limit for the number of generations is 300.
- The methods are applied to DTLZ2 with 50 distance-related variables.
- The population sizes are the same for all methods and equal 51 for $M = 2$, 136 for $M = 3$, 26 for $M = 4$, and 495 for $M = 5$.
- Apart from the interactive methods, MOEA/D is also run for the comparison.
- $L^{\boldsymbol{w}}_{\infty}$-norm preference model is used in all algorithms.
- $L^{\boldsymbol{w}^{DM}}_{\infty}$-is also used as the model of an artificial decision maker (thus, the inconsistencies should not occur). The weights are set to $\boldsymbol{w}^{DM} = [0.6, 0.4]$ for $M = 2$, $\boldsymbol{w}^{DM} = [0.6, 0.2, 0.2]$ for $M = 3$, $\boldsymbol{w}^{DM} = [0.4, 0.3, 0.2, 0.1]$ for $M = 4$, and $\boldsymbol{w}^{DM} = [0.3, 0.2, 0.2, 0.1, 0.2]$ for $M = 5$.
- *ValueModelQuality* indicator coupled with the artificial decision maker's model is employed in two forms: returning the minimal value (the best relevance score attained by




(a) $\boldsymbol{w}^{DM} = [0.5, 0.5]$

(b) $\boldsymbol{w}^{DM} = [0.25, 0.75]$

(c) $\boldsymbol{w}^{DM} = [1/3, 1/3, 1/3]$

(d) $\boldsymbol{w}^{DM} = [0.7, 0.2, 0.1]$

Figure 42: Expected final visualization (DCEMO)

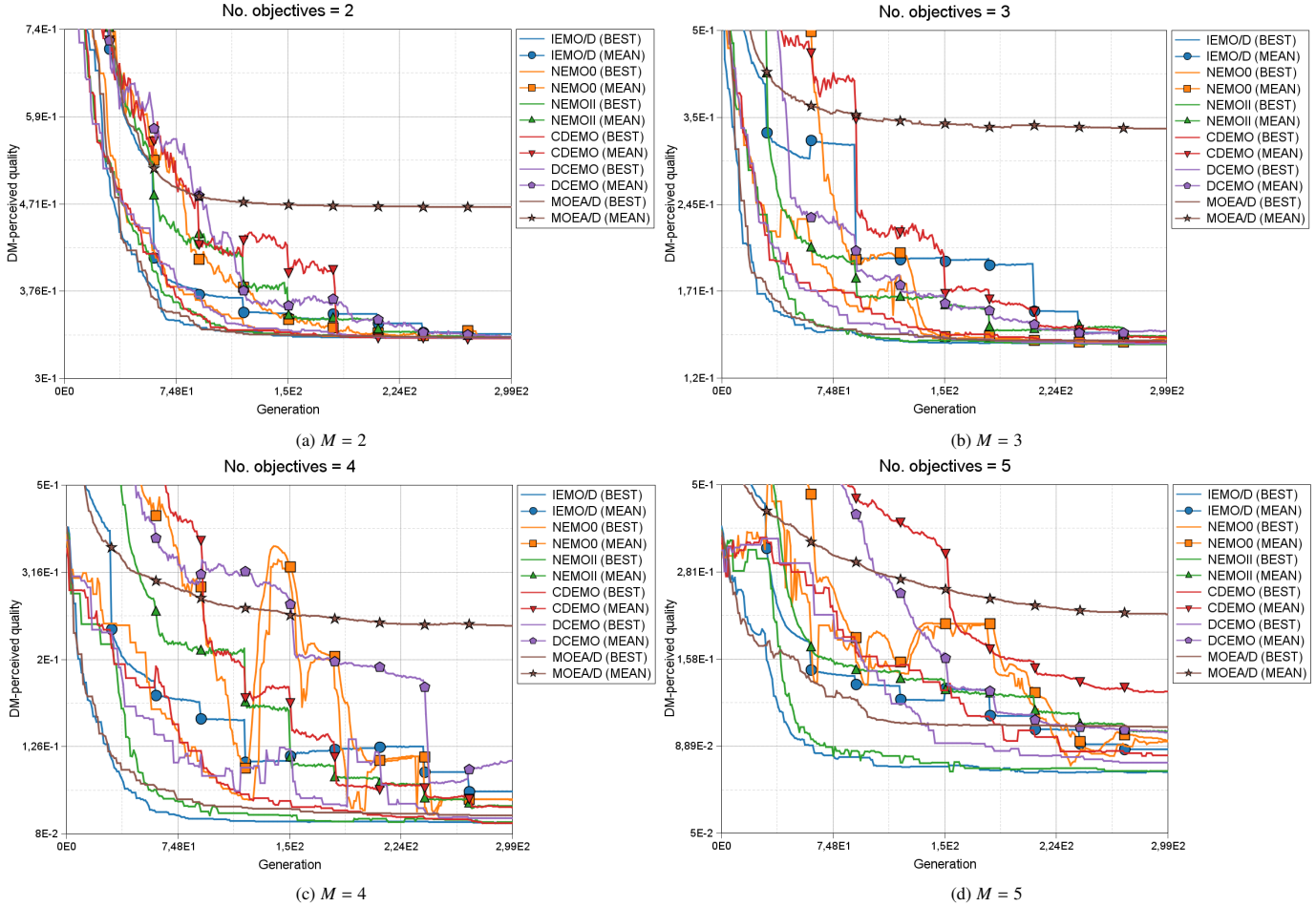
a solution in the population) and mean (average solutions' relevance).

- The results are presented as convergence plots revealing how the performance of each algorithm changed throughout all generations – according to both quality indicators employed.

The obtained results are presented in Figure 43. Although they were attained by running the methods just once per scenario (thus, their credibility can be somewhat questioned), they are consistent with the general findings on all methods. The observations that can be made are as follows:

- NEMO-0 performs highly unstable in more dimensional scenarios due to directing the search using just one representative compatible preference model instance, which may significantly change after each interaction.

- The average relevance of populations created by MOEA/D is poor – which is apparent since it approximates the Pareto front. Regarding the most relevant solution it created – it generated competitive results only in low-dimensional scenarios, where the solution sparsity is not an issue. The more objectives, however, are involved, the lesser the chances that one of the population members involved in Pareto front approximation lies near the true best decision maker's recommendation. When $M = 5$, the best solution generated by MOEA/D is even worse than the average specimen (in terms of relevance score) generated by




(a) $M = 2$

(b) $M = 3$

(c) $M = 4$

(d) $M = 5$

Figure 43: Final results

the interactive methods.

- CDEMO and DCEMO perform worse than IEMO/D and NEMO-II. This is because they are founded on an outdated evolutionary base and do not employ any mechanisms responsible for ensuring the satisfactory distribution of solutions. Among these two cone-based methods, DCEMO seems better, especially in more dimensional scenarios. The reason is that CDEMO favors cone-related criterion over non-dominated sorting, which means that the method may be more likely to not converge towards the Pareto front.

- Finally, NEMO-II and IEMO/D can be considered the best performers, with the slight advantage of IEMO/D over NEMO-II regarding the qualities of the best recommendations it constructs. It is due to being founded on a more advanced evolutionary framework.

## References


[1] J. R. F. S. Greco, M. Ehrgott, *Multiple Criteria Decision Analysis: State of the Art Surveys*, vol. 233 of *International Series in Operations Research & Management Science*. Springer, 2016.

[2] M. Kadziński, M. Ghaderi, J. Wasikowski, and N. Agell, "Expressiveness and robustness measures for the evaluation of an additive value function in multiple criteria preference disaggregation methods: An experimental analysis," *Computers & Operations Research*, vol. 87, pp. 146–164, 2017.

[3] M. K. Tomczyk and M. Kadziński, "On the elicitation of indirect preferences in interactive evolutionary multiple objective optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '20, (New York, NY, USA), pp. 569–577, ACM, 2020.

[4] S. Corrente, S. Greco, M. Kadziński, and R. Słowiński, "Robust ordinal regression in preference learning and ranking," *Machine Learning*, vol. 93, no. 2, pp. 381–422, 2013.

[5] A. D. Athanassopoulos and V. V. Podinovski, "Dominance and potential optimality in multiple criteria decision analysis with imprecise information," *The Journal of the Operational Research Society*, vol. 48, no. 2, pp. 142–150, 1997.

[6] J. Branke, S. Greco, R. Słowiński, and P. Zielniewicz, "Learning value functions in interactive evolutionary multiobjective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 1, pp. 88–102, 2015.

[7] M. K. Tomczyk and M. Kadziński, "Decomposition-based coevolutionary algorithm for interactive multiple objective optimization," *Information Sciences*, vol. 549, pp. 178–199, 2021.

[8] M. K. Tomczyk and M. Kadziński, "Decomposition-based interactive evolutionary algorithm for multiple objective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 2, pp. 320–334, 2020.

[9] M. Kadziński, M. K. Tomczyk, and R. Słowiński, "Preference-based cone contraction algorithms for interactive evolutionary multiple objective optimization," *Swarm and Evolutionary Computation*, vol. 52, p. 100602, 2020.

[10] M. Kadziński, M. K. Tomczyk, and R. Słowiński, "Preference-based cone contraction algorithms for interactive evolutionary multiple objective optimization," *Swarm and Evolutionary Computation*, vol. 52, p. 100602, 2020.