The *hMod* Software Framework

hMod is a Java-based library that includes tools, models and utilities for manually implementing algorithms, based on an *Algorithm-Assembling* object-oriented model. The main elements included within this framework are:

- An implemented algorithm structure objectual model, which include components for instantiating and assembling related entities.
- Support for managing and loading *modules*, i.e., bundles that include algorithmic components, their definitions and configurations.
- A console that allows to execute commands for experimental tasks related to algorithms.
- Scripting support (based on Javascript) for algorithm assembling and configuration.
- A bundled, lightweight, static method-based dependency injection system for algorithmic component locate purposes.

hMod combines three different perspectives. First, the objectual representation of an algorithm structure in which is based. The framework is tunned to seize such representation at different degrees. Second, hMod is suited for implementing algorithm of many types, ranging from simpler heuristics to complex hyper-heuristic architectures. And third, hMod is not tool or platform-centered. Its architecture can be used, adapted and extended to different development environments. Actually, without a bigger effort, it would be possible to port the main elements of the framework to other languages.

In this documentation, the design of hMod at the architectural level is described, alongwith examples of their usage and the development process involved.

1 Architecture

The component diagram in Figure 1 shows the general architecture of hMod. Each component can be described as follows:

- *Core*: It is the heart of the framework from the modeling perspective. It contains a set of classes and interfaces that defines an algorithm structure under the framework. The main interfaces are *Statement*, *Block*, *Condition* and *NestableExpression*. The first three are already explained in previous sections. The latter is an adaptation for any expression to support nesting, such as *Block* supports nesting for *Statement* instances. This allow to create complex conditionals that can be formed by other expressions, e.g., logic operators such as *AND-OR*. Additionally, *FlowchartFactory* provides a more larger set of factory methods for different assembling purposes.
- *Dependency manager*: Not directly a part of the framework, this component works like a dependency injection service. Through it, types can be discovered during the assembling process, which helps to ease the burden of instantiating many objects, particularly data structures. The usage of this component is based in the implementation of static methods for a particular module, and to declare dependencies and provided services on it. *ModuleLoader* is the component frontend, with which different modules are loaded to

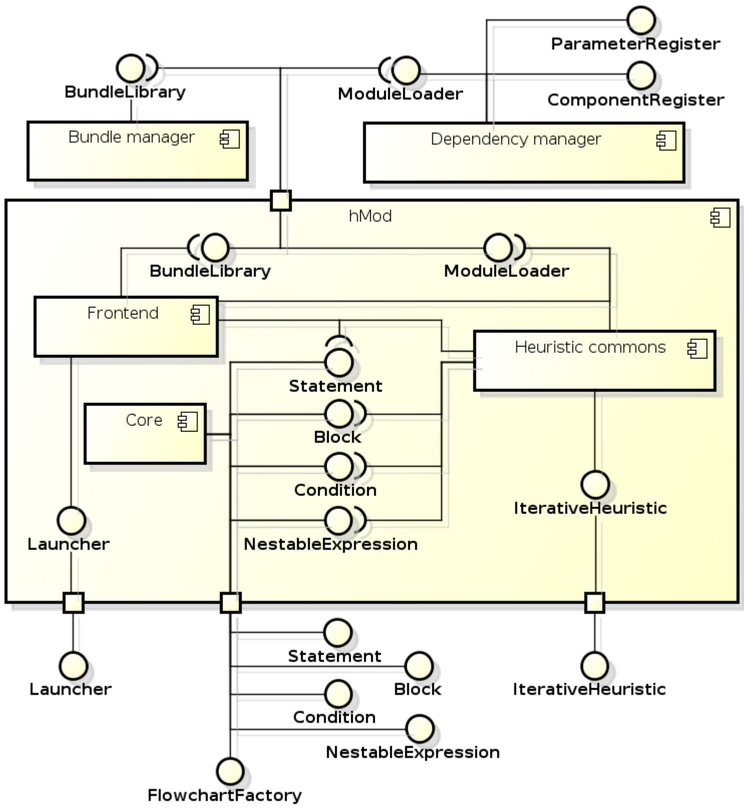


Figure 1: hMod framework architecture.

construct a single dependency tree. *ComponentRegister* and *ParameterRegister* are interfaces that allow to configure algorithm components and their runtime parameters.

- *Heuristic commons*: It is a module implementation under hMod for default heuristic support. It can be considered the most basic module that can be used on other modules for implement heuristic methods. Particularly, the *IterativeHeuristic* module implements a heuristic which can be executed a number of iterations, for which initialization and finalization procedures can be implemented.
- *Bundle manager*: Another external component that provides support to hMod features. It allows to dynamically load packed modules, i.e. *bundles*, into the development environment. It works with the *ClassLoader* Java component [5], with which modules outside the classpath can be loaded, tested and unloaded from the virtual machine (JVM) without closing the enclosing application. This provides a pretty useful environment for heuristic prototyping. *BundleLibrary* is the component frontend, with which *jar* files that contain bundles can be loaded from the file system.
- *Frontend*: It is the “tool” component of hMod. It provides the *Launcher* class, a command-like console application for performing algorithm assembling tasks. There are commands for loading bundles in pre-configured paths, configure execution interfaces, threading support, algorithm output (de)activation to screen and text files, variable manipulation, algorithm execution, among others. Additionally, scripting support is provided, based on the javascript *Nashorn* engine included in Java 8 [3]. Commands can be executed within

javascript code, giving all the benefits that such language offers. This is useful for the design and execution of complex experiments.

2 Usage tutorial: heuristics for the Dial-a-Ride Problem with Time Windows

Figure 2 shows a typical workflow for developing algorithms with hMod. It is an incremental

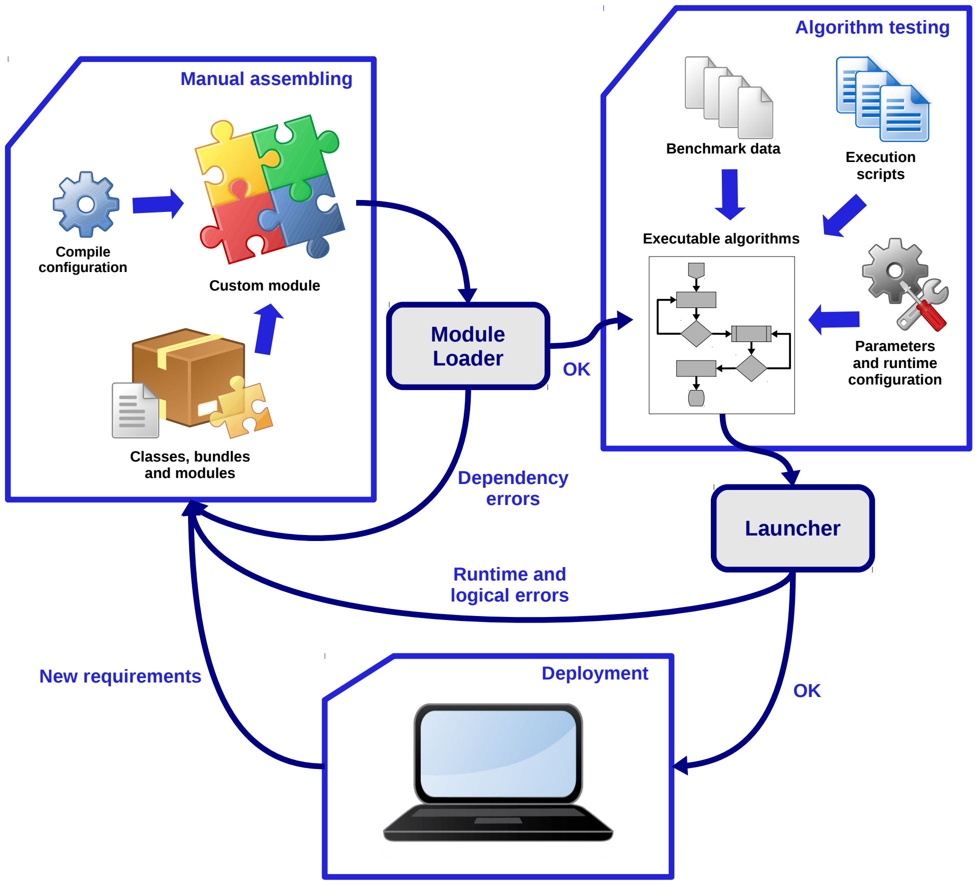


Figure 2: hMod workflow.

process for which bigger steps are divided by the use of some framework components. *Manual assembling* is the creative step in which the algorithm is implemented under the standards of the AA conceptual framework. The result is a custom module that exposes dependencies and implemented services, i.e., algorithm components. This module must be loaded through *ModuleLoader* and, if no errors arise, an executable algorithm instance is obtained. Then, in the *algorithm testing* step, the algorithm is executed through the *Launcher* component, with the support of different elements related to the problem implementation, such as benchmark data and domain parameters. If results are valid and meet the algorithm requirements, the algorithm is ready for *deploying* in its production environment. Eventually, further modifications to the initial model should be performed if requirements change.

For illustrating the use of the framework usage under this workflow, several heuristics for the Dial-a-Ride Problem with Time Windows (DARPTW) [2] will be implemented. This will be described in detail, and software artifacts related to the framework implementation will be explained in general terms.

2.1 The problem at hand

DARPTW is a transportation problem that originates from a family of pickup-and-delivery problems related to the Traveling Salesman Problem (TSP) [1]. It is known as a multi objective transportation problem, in which two relevant elements must be minimized simultaneously: the customer inconvenience (individuals are transported) and the transportation costs. A set of customers must be picked up from an origin location and they must be delivered to a destination location. For achieving this, a set of vehicles are available, and a transportation schedule must be constructed for each one, which should be subject to several constraints. In the time-window-free version of the problem (DARP), the vehicles have freedom for defining the time at which customers are picked up/delivered, but under the time-window version (DARPTW, the one considered here) a vehicle schedule must assure that the customer is served in a restricted time range: the time windows (TW) itself. That constraint adds an important complexity degree to the problem, which can be proven to be *NP*-hard [4]. In literature, there are many versions and specifications for the problem because of the variety of constraints. Here, the approach in [4] is considered, in which some complexities of the mathematical model are addressed by constraint relaxation.

More formally, there is a set of n customer transportation requests, each one associated with an pickup location i and a drop-off one $n+i$. For each request, a time window for the pickup $[a_i, b_i]$ and one for the drop-off $[a_{n+i}, b_{n+i}]$ are generated, based on some customer preferences and the system configuration. There is a set of m vehicles available for transporting the customers, each one with a fixed capacity C. Here, a *single-depot scenario* was considered, in which each vehicle k starts and ends its schedule at times T_s^k and T_e^k respectively, both at a particular depot location d. The following sets are defined:

- $P = \{1, ..., n\}$: set of pickup locations.
- $D = \{n+1, ..., 2n\}$: set of delivery locations.
- $N = P \cup D$: set of pickup and delivery locations.
- K: set of vehicles.
- $V \subset K$: set of vehicles used in solution.
- $A = N \cup \{d\}$: set of all possible stopping locations for all vehicles.

Several additional parameters are considered:

- s_i: the service time needed at location i.
- $t_{i,j}$: the traveling time or distance from location i to j.
- l_i: the change in vehicle load at location i.
- MRD: the maximum route duration, e.g., how long a vehicle schedule can be, from the starting time to the ending.

- MRT: the maximum ride time, e.g., how long the difference between the pickup and the delivery times can be for a single client.

The following decision variables are used:

- $x^k_{i,j}$: a decision variable with value 1 if the vehicle k services a customer at location i and the next customer at location j, and 0 otherwise.
- T^k_i: time at which vehicle k starts its service at location i.
- L^k_i: load of vehicle k after servicing location i.
- W^k_i: waiting time of vehicle k before servicing location i. This time is commonly generated when the vehicle arrives to i before the low-bound of the related time window.

The objective function will be weighted by using the following weights:

- w_1: weight on transport time, i.e., the total time used by vehicles for moving between locations.
- w_2: weight on excess ride time, i.e., difference between the actual time at which the customer arrives to its destination location and the time at which the customer would have reached its destination location if the vehicle transported him directly to its delivery location.
- w_3: weight on waiting time for customers, i.e., total time spent by customers onboard a stopped vehicle.
- w_4: weight on route duration, i.e., difference between the start and the end of vehicles' schedules.
- w_5: weight on time window violation, i.e., penalty applied when a vehicle arrives too early or too late at a location.
- w_6: weight on excess of maximum ride time, i.e., penalty applied when MRT is violated.
- w_7: weight on excess of route duration, i.e., penalty applied when MRD is violated.

Considering the above specifications, the mathematical model of DARPTW can be defined as follows:

$$
\begin{aligned}
\min \quad & \\
& w_1 \sum_{k\in V} \sum_{i,j\in A} t_{i,j} x^k_{i,j} \quad + \\
& w_2 \sum_{k\in V} \sum_{i\in P} (T^k_{n+i} - s_i - T^k_i - t_{i,n+i}) \quad + \\
& w_3 \sum_{k\in V} \sum_{i\in N} W^k_i (L^k_i - l_i) \quad + \\
& w_4 \sum_{k\in V} (T^k_e - T^k_s) \quad + \\
& w_5 \sum_{k\in V} \sum_{i\in A} max(0, a_i - T^k_i, T^k_i - b_i) \quad + \\
& w_6 \sum_{k\in V} \sum_{i\in P} max(0, (T^k_{n+i} - T^k_i) - MRT) \quad + \\
& w_7 \sum_{k\in V} max(0, (T^k_e - T^k_s) - MRD)
\end{aligned} \tag{1}
$$

subject to

$$\sum_{k\in V}\sum_{j\in P} x^k_{d,j} = m \tag{2}$$

$$\sum_{k\in V}\sum_{i\in D} x^k_{i,d} = m \tag{3}$$

$$\sum_{j\in A} x^k_{i,j} - \sum_{j\in A} x^k_{j,i} = 0 \qquad \forall k \in V, i \in N \tag{4}$$

$$\sum_{k\in V}\sum_{j\in N} x^k_{i,j} = 1 \qquad \forall i \in P \tag{5}$$

$$\sum_{j\in N} x^k_{i,j} - \sum_{j\in N} x^k_{j,n+i} = 0 \qquad \forall k \in V, i \in P \tag{6}$$

$$x^k_{i,j}(T^k_i + s_i + t_{i,j} + W^k_j - T^k_j) \leqslant 0 \quad \forall k \in V, i, j \in A \tag{7}$$

$$T^k_i + s_i + t_{i,n+i} + W^k_j - T^k_{i+n} \leqslant 0 \qquad \forall k \in V, i \in P \tag{8}$$

$$x^k_{i,j}(L^k_i + l_j - L^k_j) = 0 \qquad \forall k \in V, \quad i, j \in A \tag{9}$$

$$l_i \leqslant L^k_i \leqslant C \qquad \forall k \in V, i \in P \tag{10}$$

$$L^k_d = 0 \qquad \forall k \in V \tag{11}$$

$$x^k_{i,j} \in \{0,1\} \qquad \forall k \in K, \quad i, j \in A \tag{12}$$

$$T^k_i \geqslant 0 \qquad \forall k \in K, \quad i \in V \tag{13}$$

$$L^k_i \geqslant 0 \qquad \forall k \in K, \quad i \in V \tag{14}$$

$$W^k_i \geqslant 0 \qquad \forall k \in K, \quad i \in V \tag{15}$$

The constraints defined above can be classified according to the following criteria:

- *Depot constraints*: (2) and (3) forces a vehicle for starting and finishing its schedule in the depot location.
- *Routing constraints*: (4) ensures that, for each location, there is an equally number of vehicles arriving and leaving. (5) ensures that exactly one vehicle serves each pickup location, and (6) forces the same vehicle to serve both the pickup and delivery locations, for any customer.
- *Precedence constraints*: (7) ensures that the arrival time at any location is greater than the leaving time of the previous location in the route. (8) ensures that the pickup location of any customer is visited before its delivery location.
- *Vehicle load constraints*: (9) forces that, for each vehicle, the same quantity of pickups and deliveries are performed within a route, (10) ensures that vehicle capacity is never exceeded and (11) forces that all vehicles must be empty when starting and ending their schedules.

Figure 3 shows a high-level class diagram of the DARPTW domain classes used for this example. As they do not directly relate to assembling functionalities, further details were omitted. Note that this model has nothing to do with hMod standards. Actually, it could be provided as an external package or bundle. Then, assembling components could be implemented on top of it. *DARPTWFactory* is the model frontend, which allows to create a new problem instance, the *ProblemInstance* class, from a specific benchmark file. With a *ProblemInstance* reference it is possible to create empty routes, implemented through the *Route* class. Routes are the most important entities for solution building. Also, with a *ProblemInstance* reference, it is possible to create an *Evaluator*, which allows to process and convert a set of *Route* instances into a concrete solution of the problem, a *DARPTWSolution* instance. The latter stores all calculations

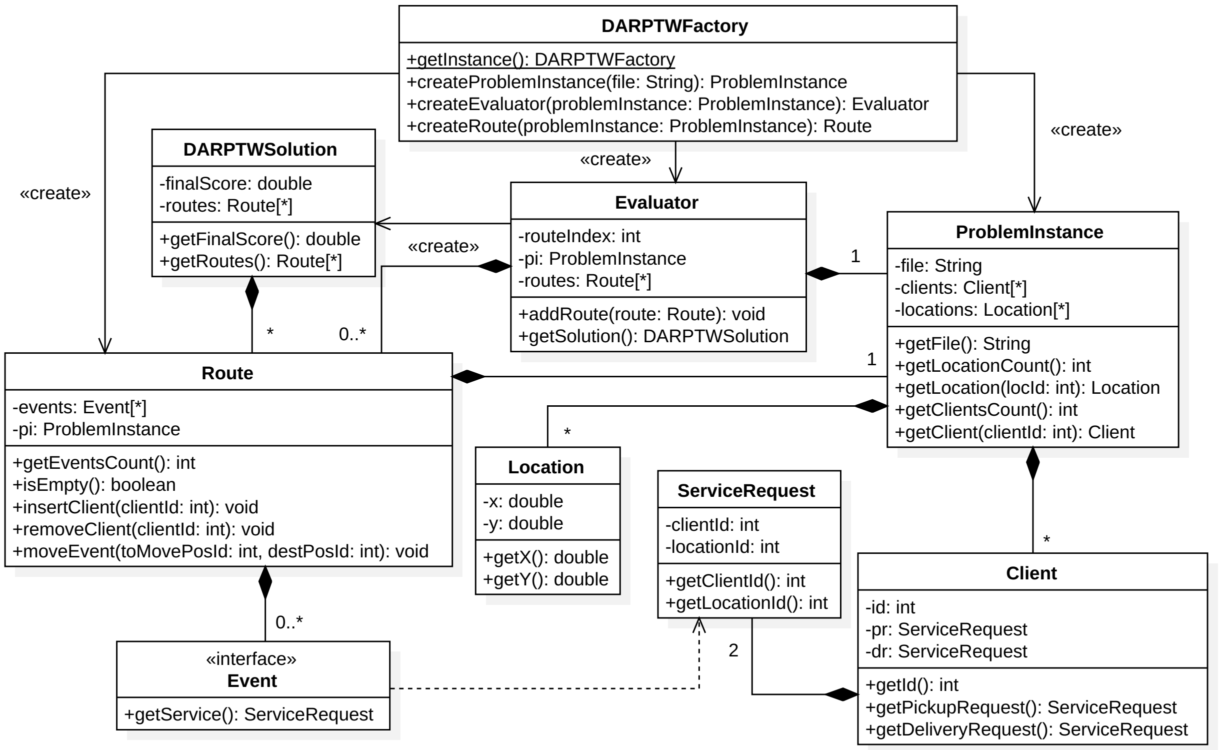


Figure 3: A high-level class diagram of DARPTW entities.

related to the objective function, obtained from the *Evaluator* itself, an based on the configured system weights. A route contains *Events*, i.e., entities that represent an action within the route schedule, such as pickup and delivery. Internal implementations of this interface work like a double-chained linked list, but them are actually encapsulated within the package (and not showed in this model). Finally, a single *ProblemInstance* stores data about *Locations*, i.e., cartesian coordinates of the transit map, and *Clients*, i.e., customers. The latter additionally store two *ServiceRequest* instances, which save information regarding customer transportation requirements for pickup and delivery.

2.2 Assembling an initializer heuristic

The heuristics to implement should be defined at this point. As this is just the start, an initializer heuristic will be addressed. Consider a procedure in which all available clients will be inserted in a random route within the solution. A general pseudocode for this heuristic is shown in Listing 1. Two groups of components can start to be identified in it: data structures and operations. There is no concrete recipe to define this information precisely, it is a design matter. Starting components can be defined at first, and they can be further complemented and updated with new features through an iterative approach. By standard, there is a two-level classification for them:

- Regarding *assembling utility*:
  - *Algorithm-Assembling based components*: This considers components related to data structures, elemental operators, etc.

```
foreach available client $c$ in solution
    select a random route $r$ from the solution
    select a random and feasible pickup position $p$ in the current schedule of $r$
    select a random and feasible delivery position $d$ in the current schedule of $r$
    insert client $c$ pickup event at position $p$ in $r$
    insert client $c$ delivery event at position $d$ in $r$
end for
```

Listing 1: A pseudocode for a random solution initialization in DARPTW.

    - *General purpose-components*: other components non considered in the previous point, e.g., benchmark data parsers, entity classes and domain collections that only provide supporting functions to algorithmic components.

- Regarding *algorithmic scope*:

    - *Domain components*: they were implemented for addressing requirements of a specific problem domain or context, e.g., classes directly related to DARPTW or MKP.
    - *Solver components*: they were implemented to provide functionality related to a specific algorithm or heuristic method, e.g., classes related to a GA or a hyper-heuristic implementation.

Eventually, a single component may be classified in more than one sub-category, but it should be predominant in only one of the above described. For the example in this section, only domain components will be implemented in the algorithmic scope category. Because this, a single package will be used to group all components of the domain. In hMod, the standard is to use a `hmod.domains.`*`mydomain`* package, in this case, `hmod.domains.darptw` (the same for the domain entities). When implementing solver components, the standard is to use a `hmod.solvers.`*`mysolver`* package. Note that further nesting of specific domain or solver models can be done within one of these packages.

For the heuristic in Listing 1, the following data can be considered:

- The *ProblemInstance* related for the loaded benchmark, for getting information about clients and their requirements.
- An entity that helps to build a solution by storing the current routes. It will be named *SolutionBuilder*.
- An entity through which concrete solutions instances (*DARTWSolution*) can be saved and loaded from the constructed routes. It will be named *SolutionHandler*.
- An entity that stores a mutable list of clients. The *List* interface of the `java.util` package can be used.
- An entity that stores a selected route. It will be named *RouteHandler*.
- An entity that stores a selected position for a client within a route. It will be named *InsertionPositionHandler*. It includes both the pickup and delivery positions.
- An entity that stores a selected client. It will be named *ClientHandler*.
- A structure for controlling the iteration will be necessary. hMod incorporates in the heuristic commons package (`hmod.solvers.common`) the *IterationHandler* class for such purposes.

Regarding operators, the class *DARPTWProcesses* will provide a set of factory methods for *Statement* instances. Such methods will be the following:

- `fillAvailableClients`: Takes the available clients from a *SolutionBuilder* instance and store them in a *List* instance.
- `initIterationForClients`: Initializes an *IterationHandler* instance by using the number of clients in a *List* instance.
- `selectClientFromIterator`: Using the current iteration number from an *IterationHandler* instance, takes a client from a *List* instance and stores it into a *ClientHandler* instance.
- `selectRandomRoute`: Gets a random route from a *SolutionBuilder* instance and stores it into a *RouteHandler* instance.
- `selectRandomInsertionPointInRoute`: Selects a random feasible pickup and delivery position from a route in a *RouteHandler* instance, and stores them into a *InsertionPositionHandler* instance.
- `insertClient`: Inserts the client in a *ClientHandler* instance within the route in a *RouteHandler* instance, on the insertion positions in a *InsertionPositionHandler* instance.
- `reportResult`: Prints the description of an output solution in a *SolutionHandler* instance.

Other required operations may be provided through the interfaces of data structures directly. Listing 2 shows the code of the `fillAvailableClients` method. Note that in line 5, a

```
 1  public static Statement fillAvailableClients(SolutionBuilder sb,
 2                                              ProblemInstance pi,
 3                                              List<Client> targetList)
 4  {
 5      return () -> {
 6          BitSet availables = sb.getAvailableClients();
 7          int clientId = availables.nextClearBit(0);
 8
 9          while(clientId < pi.getClientsCount())
10          {
11              targetList.add(pi.getClient(clientId + 1));
12              clientId = availables.nextClearBit(clientId + 1);
13          }
14      };
15  }
```

Listing 2: Example of how a factory method can provide a simple operator in the *DARPTWProcesses* class.

lambda expression syntax of Java 8 is used to create the *Statement* instance. This is possible because, as hMod provides it, the *Statement* type can be treated as a functional interface (only has the *run()* method). Other operators may be more simpler or complex than the presented one. When an operator code is getting bigger than expected, it may be adequate to separate it into multiple operators, and after they can be mixed in the algorithm structure. This make them not only more manageable but more reusable.

Listing 3 shows a test unit through which all the above components are mixed into an executable algorithm structure to finally construct the heuristic. Several actions related to the prob-

```java
 1 import static hmod.core.FlowchartFactory.NOT;
 2 import static hmod.core.FlowchartFactory.While;
 3 import static hmod.core.FlowchartFactory.block;
 4 // ...
 5 public class DARPTWTests
 6 {
 7     // ...
 8     @Test
 9     public void standaloneTest()
10     {
11         // Instance and weights initialization
12         ProblemInstance pi = DARPTWFactory.getInstance().createProblemInstance("benchmark-file");
13 
14         MutableFactorMap weightsMap = new MutableFactorMap();
15         weightsMap.addFactor(Factor.TRANSIT_TIME, FactorValue.create(8.0));
16         weightsMap.addFactor(Factor.ROUTE_DURATION, FactorValue.create(1.0));
17         weightsMap.addFactor(Factor.SLACK_TIME, FactorValue.create(0.0));
18         weightsMap.addFactor(Factor.RIDE_TIME, FactorValue.create(0.0));
19         weightsMap.addFactor(Factor.EXCESS_RIDE_TIME, FactorValue.create(3.0));
20         weightsMap.addFactor(Factor.WAIT_TIME, FactorValue.create(1.0));
21         weightsMap.addFactor(Factor.TIME_WINDOWS_VIOLATION, FactorValue.create(pi.getClientsCount()));
22         weightsMap.addFactor(Factor.MAXIMUM_RIDE_TIME_VIOLATION, FactorValue.create(pi.getClientsCount()));
23         weightsMap.addFactor(Factor.MAXIMUM_ROUTE_DURATION_VIOLATION, FactorValue.create(pi.getClientsCount()));
24 
25         // Data initialization
26         List<Client> clients = new ArrayList<>();
27         MutableIterationHandler iterator = new MutableIterationHandler();
28         ClientHandler client = new ClientHandler();
29         RouteHandler route = new RouteHandler();
30         InsertionPositionHandler pos = new InsertionPositionHandler();
31         SolutionBuilder sb = new SolutionBuilder(pi);
32         SolutionHandler sh = new SolutionHandler(pi, sb, weightsMap);
33 
34         // Assembling
35         Statement heuristic = block(
36             sh::loadEmptySolution,
37             clients::clear,
38             DARPTWProcesses.fillAvailableClients(sb, pi, clients),
39             DARPTWProcesses.initIterationForClients(iterator, clients),
40             While(NOT(iterator::areIterationsFinished)).Do(
41                 DARPTWProcesses.selectClientFromIterator(iterator, clients, client),
42                 DARPTWProcesses.selectRandomRoute(sb, route),
43                 DARPTWProcesses.selectRandomInsertionPointInRoute(route, pos),
44                 DARPTWProcesses.insertClient(pos, route, client),
45                 iterator::advanceIteration
46             ),
47             sh::saveSolutionToOutput,
48             DARPTWProcesses.reportResult(sh)
49         );
50 
51         // Output config
52         OutputManager.getCurrent().setOutputsFromConfig(new OutputConfig().
53             addSystemOutputId(DARPTWOutputIds.RESULT_DETAIL)
54         );
55 
56         // Execution
57         heuristic.run();
58     }
59     // ...
60 }
```

Listing 3: Test example of how to manually instantiate algorithms components and arrange them into the structure in hMod.

lem implementation are required before performing the assembling. At line 12, the problem instance is created through the *DARPTWFactory* class. Between lines 14 and 23, a configuration for the weights in objective function is created through the *MutableFactorMap* class. It is a simple collection for storing these weights, and is used for solution evaluation purposes. Between lines 26 and 32, all relevant data structures are directly created. Note that several dependencies exist between them. From line 35 to 49, the assembling is actually done. This is the part of the code in where framework tools for creating an objectual model are used. Note that, at the start of the class, static imports are declared to directly use the factory methods defined in the *FlowchartFactory* class. hMod includes the *block()* method for arranging a set of operators through array of *Statement* instances. There are two main ways through which such instances can be passed to the *block()* call:

- If a data structure exposes in its interface a *Statement*-like method, i.e., a method with no arguments and return type, then it can be directly put in the structure by using a *method reference*. It is a Java 8 feature that allows to convert methods which follow a specific signature to a functional interface instance without the need of create additional classes, including anonymous ones. The expression `sh::loadEmptySolution` corresponds to a method reference, considering that the *SolutionHandler* class exposes the method `void loadEmptySolution()`. It is rather equivalent to use the lambda expression `() -> sh.loadEmptySolution()`.
- If an operator requires more data than the structure in which it is defined, then it is better to define a factory method that return a *Statement* instance and receives by argument all data structures required to perform the operation. Calls like `DARPTWProcesses.fillAvailableClients(sb,pi,clients)` represent this approach.

Finally, note that a *Condition* is used in line 40 within a while structure. A method reference is used as well, because the method *areIterationFinished()* follows a condition signature. The *NOT()* method creates a negated version of the condition passed as argument.

The created structure is stored in the `st` variable. Before running it, *algorithms outputs* must be configured to grab results from the execution. hMod includes a simple output managing system, which can be referenced within operators to provide printing towards screen or files. The concept of this system is that operations provide outputs *without specifying the means* through which they will be showed to the user. Rather, execution configuration will be the one that specifies if the output is printed to the screen, to a file, or whatever. Consider the code of the `reportResult` operator in Listing 4. In line 8, the *OutputManager*

```
 1
 2  public static Statement reportResult(SolutionHandler sh)
 3  {
 4      return () -> {
 5          DARPTWSolution best = sh.getBestSolution();
 6          FactorMap eval = best.getEvaluation();
 7
 8          OutputManager.println(DARPTWOutputIds.RESULT_DETAIL, "Best solution found:\n" +
                best);
 9          OutputManager.println(DARPTWOutputIds.RESULT_SHEET,
10              eval.getFactorValue(Factor.TRANSIT_TIME) + "\t" +
11              eval.getFactorValue(Factor.ROUTE_DURATION) + "\t" +
12              eval.getFactorValue(Factor.SLACK_TIME) + "\t" +
13              eval.getFactorValue(Factor.RIDE_TIME) + "\t" +
14              eval.getFactorValue(Factor.EXCESS_RIDE_TIME) + "\t" +
15              eval.getFactorValue(Factor.WAIT_TIME) + "\t" +
16              eval.getFactorValue(Factor.TIME_WINDOWS_VIOLATION) + "\t" +
17              eval.getFactorValue(Factor.MAXIMUM_ROUTE_DURATION_VIOLATION) + "\t" +
18              eval.getFactorValue(Factor.MAXIMUM_RIDE_TIME_VIOLATION)
19          );
20      };
21  }
```

Listing 4: Example of how to support output within an operator.

class is used for printing through a pre-configured output. The *println()* method, receives two arguments: the first is a string for an *output identifier*, and the second is the string to be printed. *OutputManager* checks if there is an output configured with such id and, if so, prints the message through the associated outputs means. If no output was configured, the message is simply discarded. This operator make two output calls, one for printing the details of the resultant solution (the `DARPTWOutputIds.RESULT_DETAIL` constant), and one for printing in a tabular format the individual scores of different objective function factors (the `DARPTWOutputIds.RESULT_SHEET` constant). The latter is an example of an output that can be configured to be flushed into a file. Getting back to the Listing 3, in line 52, the `DARPTWOutputIds.RESULT_DETAIL` output is configured to be flushed towards system output, i.e., console. The *OutputConfig* class allows to configure this and other output means within the same id.

Finally, the line 57 in Listing 3 runs the heuristic, which is the same that running the generated *Statement* instance. The output of the execution is shown in Listing 5.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running hmod.domains.darptw.DARPTWTests
Best solution found:
Solution score: 253783.3099947404 (Unfeasible)
Total Transit time: 345.4
Total Route duration: 1010.8
Total Slack time: 185.5
Total Ride time: 1679.6
Total Excess ride time: 1528.1
Total Wait time: 289.7
Total Time windows violation: 9642.1
Total Maximum route duration violation: 100.1
Total Maximum ride time violation: 471.7

*** Route 1 *** (clients: 11; Transit time: 174.7; Route duration: 580.1; Slack time: 185.5; Ride time: 1058.7; Excess
      ride time: 983.5; Wait time: 289.7; Time windows violation: 5499.7; Maximum route duration violation: 100.1;
      Maximum ride time violation: 391.8)
<D0> (5) (6) (-6) (19) (12) (16) (-5) (9) (14) (-12) (-9) (11) (-14) (10) (-11) (-16) (-19) (22) (-22) (-10) (21) (-21)
      <DN>

*** Route 2 *** (clients: 2; Transit time: 28.7; Route duration: 68.7; Slack time: 0; Ride time: 12.8; Excess ride time:
       0; Wait time: 0; Time windows violation: 24.6; Maximum route duration violation: 0; Maximum ride time violation:
      0)
<D0> (18) (-18) (15) (-15) <DN>

*** Route 3 *** (clients: 11; Transit time: 142; Route duration: 362; Slack time: 0; Ride time: 608.1; Excess ride time:
       544.7; Wait time: 0; Time windows violation: 4117.8; Maximum route duration violation: 0; Maximum ride time
      violation: 80)
<D0> (13) (3) (-13) (1) (24) (-24) (4) (-3) (7) (-4) (17) (23) (-17) (-1) (20) (-7) (-20) (2) (-23) (8) (-2) (-8) <DN>

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.356 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Listing 5: Output extract of the test in Listing 3.

2.3 Reusing components

With the components implemented for the initializer heuristic, plus new components that can be added for specific purposes, it is possible to assemble new heuristics for the problem. Consider now the pseudocode in Listing 6. It corresponds to a simple heuristic that moves a customer from one route to another in the solution. All is selected at random, the customer and the destination route. The reader may notice several similarities with the pseudocode in Listing 1. Precisely, the idea is to exploit these similarities by reusing operators and data structures in this new heuristic.

```
select a random route $r_s$ from the solution
select a random route $r_s$ from the solution, different from $r_d$
select a random client $c$ currently assigned to route $r_s$
select a random and feasible pickup position $p$ in the current schedule of $r_d$
select a random and feasible delivery position $d$ in the current schedule of $r_d$
remove client $c$ from $r_s$
insert client $c$ pickup event at position $p$ in $r_d$
insert client $c$ delivery event at position $d$ in $r_d$
```

Listing 6: A pseudocode of a heuristic for moving a random client.

First, a new data structure will be added: the *RouteSet* class. It represents a simple set of routes that can be include and exclude *Route* instances for different purposes. As this is partially performed by *SolutionBuilder*, this class will be changed to extend *RouteSet*. Then, the following new operators will be added to the *DARPTWProcesses* class:

- `pickModifiableRoutes`: Takes the routes from a *RouteSet* instance that can be modified, i.e., that has at least one client, and store them in another *RouteSet* instance.
- `selectOtherRoute`: Select a random route from a *RouteSet* instance, different from the one stored in a *RouteHandler* instance, and stores it into another *RouteHandler* instance.
- `removeClient`: Removes the client stored in a *ClientHandler* instance from the route in a *RouteHandler* instance.
- `moveClient`: executes sequentially `removeClient` and `insertClient`, having the origin route in a *RouteHandler* instance, the destination route in a *RouteHandler* instance, the client to move in a *ClientHandler* instance and the pickup and delivery target positions of the target route in a *InsertionPositionHandler* instance.

Before implementing the new heuristic, a small issue must be considered: it requires an already constructed solution. At this point, it will be nice to reuse the initializer heuristic implemented before for this purpose. The test class of Listing 3 will be refactored to extract this heuristic into a factory method within the same class. Listing 7 shows this method properly implemented. Note that it receives the *SolutionBuilder* and *ProblemInstance* instances used in the main code, as them are shared across different heuristics.

Now, the new heuristic can be implemented within the class, by using the same factory method approach. The result is showed in Listing 8, alongwith the modification within the test method that allows to use it.

2.4 Packaging the module

Now there are two functional heuristics for DARPTW implemented, but this is only at test level. For a more definitive approach, they should be packed within a *module*. In hMod terms, a module is a component that offers algorithm solutions already assembled, which may depend at the same time on algorithmic components that can reside in different modules. The module frontend should be presented as a class whose interface exposes the executable entities related to the solutions provided. In terms of integration, a module could be considered a hMod *plug-in*, because it should be (un)loadable by the *BundleLibrary* component.

```java
private Statement initHeuristic(SolutionBuilder sb, ProblemInstance pi)
{
    List<Client> clients = new ArrayList<>();
    MutableIterationHandler iterator = new MutableIterationHandler();
    ClientHandler client = new ClientHandler();
    RouteHandler route = new RouteHandler();
    InsertionPositionHandler pos = new InsertionPositionHandler();

    return block(
        clients::clear,
        DARPTWProcesses.fillAvailableClients(sb, pi, clients),
        DARPTWProcesses.initIterationForClients(iterator, clients),
        While(NOT(iterator::areIterationsFinished)).Do(
            DARPTWProcesses.selectClientFromIterator(iterator, clients, client),
            DARPTWProcesses.selectRandomRoute(sb, route),
            DARPTWProcesses.selectRandomInsertionPointInRoute(route, pos),
            DARPTWProcesses.insertClient(pos, route, client),
            iterator::advanceIteration
        )
    );
}

@Test
public void standaloneTest()
{
    //...

    SolutionBuilder sb = new SolutionBuilder(pi);
    SolutionHandler sh = new SolutionHandler(pi, sb, weightsMap);

    Statement heuristic = block(
        sh::loadEmptySolution,
        initHeuristic(sb, pi),
        sh::saveSolutionToOutput,
        DARPTWProcesses.reportResult(sh)
    );

    //...
}
```

Listing 7: The initializer heuristic of Listing 3 refactored to a factory method.

The first step for creating the module is to create its frontend class. For specific problem domains, it is common to use the *Domain* suffix in the name, therefore, for DARPTW the frontend class will be named *DARPTWDomain*, and it should be put into the `hmod.domains.darptw` package. A first skeleton of the module could be as Listing 9 shows. There are a few things to notice regarding the implementation done in tests. First, blocks are created sightly different, only to not create conflicts with local data structures used on each heuristic. The call used is something like `block(() -> ... return block(...); )`, which allows to create through a lambda expression local data structure variables when the block is assembled. Second, some new and simpler *Statement* instances are exposed, such as `loadNewSolution` and `reportSolution`. Also, a *Statement* getter for each heuristic is provided by the module class.

With the current implementation, the *DARPTWDomain* class is not more than a common class which can be instantiated through its constructor. It exposes its dependencies through the constructor itself: there are several data structures declared there. This forces the clients of the class to learn about how these dependencies must be created before creating the module itself.

```java
private Statement moveRandomClient(SolutionBuilder sb, ProblemInstance pi)
{
    RouteSet modRoutes = new RouteSet();
    RouteHandler sourceRoute = new RouteHandler();
    RouteHandler targetRoute = new RouteHandler();
    ClientHandler toMove = new ClientHandler();
    InsertionPositionHandler pos = new InsertionPositionHandler();

    return block(
        modRoutes::clear,
        DARPTWProcesses.pickModifiableRoutes(sb, modRoutes),
        DARPTWProcesses.selectRandomRoute(modRoutes, sourceRoute),
        DARPTWProcesses.selectOtherRoute(modRoutes, sourceRoute, targetRoute),
        DARPTWProcesses.selectRandomClientFromRoute(pi, sourceRoute, toMove),
        DARPTWProcesses.selectRandomInsertionPointInRoute(targetRoute, pos),
        DARPTWProcesses.moveClient(sourceRoute, targetRoute, toMove, pos)
    );
}

@Test
public void standaloneTest()
{
    //..

    Statement heuristic = block(
        sh::loadEmptySolution,
        initHeuristic(sb, pi),
        moveRandomClient(sb, pi),
        sh::saveSolutionToOutput,
        DARPTWProcesses.reportResult(sh)
    );

    //..
}
```

Listing 8: Implementation of the heuristic for moving a random client, integrated with the initialization heuristic.

The problem is that such a task can be very complex when dependencies are complex to address at the same time. Because this, hMod enforces the use of the *ModuleLoader* component to avoid this complexity in some degree. To make the *DARPTWDomain* class compatible to the module loading system, it is necessary to i) hide the public constructor, and ii) provide a special *static loader method* in the same class (or other in package) for loading purposes. The main advantage of this method is that not only the module itself but all related dependencies can be provided at the same time in the same module.

A loader method should be static, have no return type and a number of arguments which define the module dependencies. Listing 10 shows a pretty simple loader method for the *DARPTWDomain* class. There are several remarks in such code. First, a loader method should indicate what components are provided, and this must be done through the *LoadsComponent* annotation (see line 3). A list of *Class* instances, i.e., the provided types, must be specified in the annotation. Then, a call to the *provide()* method in the *ComponentRegister* class must be performed before finishing the method for each provided class. This will satisfy the service declared as provided. Components can be provided by using only a public type instance, or by using a private or package-protected type instance along with a public type. If some class is declared in the *LoadsComponent* annotation and is not provided in the method body, loading errors will arise. Finally, it is important to notice that the constructor of *DARPTWDomain* is now private. This

```java
public final class DARPTWDomain
{
    private final Statement loadNewSolution;
    private final Statement saveSolution;
    private final Statement reportSolution;
    private final Statement initHeuristic;
    private final Statement moveRandomClientHeuristic;

    public DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
    {
        initHeuristic = block(() -> {
            List<Client> clients = new ArrayList<>();
            MutableIterationHandler iterator = new MutableIterationHandler();
            ClientHandler client = new ClientHandler();
            RouteHandler route = new RouteHandler();
            InsertionPositionHandler pos = new InsertionPositionHandler();

            return block(
                clients::clear,
                DARPTWProcesses.fillAvailableClients(sb, pi, clients),
                DARPTWProcesses.initIterationForClients(iterator, clients),
                While(NOT(iterator::areIterationsFinished)).Do(
                    DARPTWProcesses.selectClientFromIterator(iterator, clients, client),
                    DARPTWProcesses.selectRandomRoute(sb, route),
                    DARPTWProcesses.selectRandomInsertionPointInRoute(route, pos),
                    DARPTWProcesses.insertClient(pos, route, client),
                    iterator::advanceIteration
                )
            );
        });

        moveRandomClientHeuristic = block(() -> {
            RouteSet modRoutes = new RouteSet();
            RouteHandler sourceRoute = new RouteHandler();
            RouteHandler targetRoute = new RouteHandler();
            ClientHandler toMove = new ClientHandler();
            InsertionPositionHandler pos = new InsertionPositionHandler();

            return block(
                modRoutes::clear,
                DARPTWProcesses.pickModifiableRoutes(sb, modRoutes),
                DARPTWProcesses.selectRandomRoute(modRoutes, sourceRoute),
                DARPTWProcesses.selectOtherRoute(modRoutes, sourceRoute, targetRoute),
                DARPTWProcesses.selectRandomClientFromRoute(pi, sourceRoute, toMove),
                DARPTWProcesses.selectRandomInsertionPointInRoute(targetRoute, pos),
                DARPTWProcesses.moveClient(sourceRoute, targetRoute, toMove, pos)
            );
        });

        loadNewSolution = block(
            sh::loadEmptySolution,
            initHeuristic
        );

        saveSolution = sh::saveSolutionToOutput;
        reportSolution = DARPTWProcesses.reportResult(sh);
    }

    public Statement loadNewSolution() { return loadNewSolution; }
    public Statement saveSolution() { return saveSolution; }
    public Statement reportSolution() { return reportSolution; }
    public Statement initHeuristic() { return initHeuristic; }
    public Statement moveRandomClientHeuristic() { return moveRandomClientHeuristic; }
}
```

Listing 9: A first skeleton for a module of the DARPTW problem.

allows to control the creation of the module through the loader method only.

With the described change, it is possible to test the *ModuleLoader* class. Listing 11 shows a test method for this purpose. First, an instance of *ModuleLoader* is created. The *loadAll()* call registers loader methods implemented in the classes used as argument. If more than one loader method resides in a provided class, all of them are loaded as well. It is also possible to load only specific methods with the *load()* method. In this case, it is only needed to load the *DARPTWDomain* class. Then, the *getInstance()* method allows to resolve the dependencies and to create an instance of the requested component. In this case, a *DARPTWDomain* instance is created, and it is used after for creating the same block structure than the code in Listing 8.

If the test method is executed, a dependency error like the following will be thrown:

```
optefx.loader.UnmetDependencyException:  The requested dependency (class ProblemInstance) by
the loader 'public static void DARPTWDomain.loadDomain(ComponentRegister, ProblemInstance,
SolutionHandler, SolutionBuilder)' is not registered
```

This actually indicates that there are declared dependencies that are not provided within the loading process, and this is obvious: there is no place in which the *ProblemInstance* instance is created. There are two options for this: to use an external module that provides such component, or to implement an own loader method which provides it. The latter one will be used. A new loader method is provided in Listing 12. Such code could be added to the already implemented loader method, but the idea is to illustrate the use of multiple loader methods. Note that creating the problem instance requires a file path. For this, the *Parameter* type is introduced in line 3. It allows to create required and optional parameter identifiers for the creation of a specific component. They should be defined as constants: by using the `public`, `static` and `final` modifiers. They can also receive a name for debugging purposes (in this case, "`DARPTWDomain.INSTANCE`"). Within the loader method, to access configured parameter values, a *ParameterRegister* instance can be included as a method argument. The value set for the parameter is obtained in line 8, by using the *getRequiredValue()* method of *ParameterRegister*. This force the parameter to be set, or an exception is thrown.

To finish this example, we need to provide the remaining components that are required by the *DARPTWDomain* class. This includes a *SolutionBuilder* and a *SolutionHandler* instance. Particularly, the latter requires weight factors of the objective function to perform solution evaluations. This can be addressed in a straightforward manner with additional parameters. Listing 13 shows the final version of the module with these changes included in a new loader method named *loadSolutionData()*. Note that this method declares a dependency to a *ProblemInstance*, which is used to create *SolutionBuilder* and *SolutionHandler*. The next step is to update the test method to include the parameters added to the module. Listing 14 shows this updated code, whose execution now is successful. To use the defined parameters, the *setParameter()* method of the *ModuleLoader* class must be used, by providing the parameter to set and its value. When dependencies are resolved, the *ModuleLoader* components automatically searches for registered load methods, and examine their dependencies and provided services. For this example, the execution order that satisfies the declared dependencies is *loadProblemInstance()*, *loadSolutionData()* and *loadDomain()*. Complex module dependencies can be automatically solved by *ModuleLoader*, liberating the developer to create extensive initialization routines.

```java
public final class DARPTWDomain
{
    @LoadsComponent(DARPTWDomain.class)
    public static void loadDomain(ComponentRegister cr,
                                  ProblemInstance pi,
                                  SolutionHandler sh,
                                  SolutionBuilder sb)
    {
        cr.provide(new DARPTWDomain(sh, pi, sb));
    }

    //...

    private DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
    {
        //...
    }

    //...
}
```

Listing 10: A simple module loader method.

```java
@Test
public void moduleTest()
{
    DARPTWDomain domain = new ModuleLoader().
        loadAll(DARPTWDomain.class).
        getInstance(DARPTWDomain.class);

    Statement heuristic = block(
        domain.loadNewSolution(), // initHeuristic is included in loadNewSolution()!
        domain.moveRandomClientHeuristic(),
        domain.saveSolution(),
        domain.reportSolution()
    );

    OutputManager.getCurrent().setOutputsFromConfig(new OutputConfig().
        addSystemOutputId(DARPTWOutputIds.RESULT_DETAIL)
    );

    heuristic.run();
}
```

Listing 11: A test method in which *ModuleLoader* is used.

```java
public final class DARPTWDomain
{
    public static final Parameter<String> INSTANCE = new Parameter<>("DARPTWDomain.INSTANCE");

    @LoadsComponent(ProblemInstance.class)
    public static void loadProblemInstance(ComponentRegister cr, ParameterRegister pr)
    {
        String file = pr.getRequiredValue(INSTANCE);
        cr.provide(DARPTWFactory.getInstance().createProblemInstance(file));
    }

    //...
}
```

Listing 12: Example of a loader method that uses parameters.

```java
public final class DARPTWDomain
{
    public static final Parameter<String> INSTANCE = new Parameter<>("DARPTWDomain.INSTANCE");
    public static final Parameter<Double> WEIGHT_TRANSIT_TIME = new Parameter<>("DARPTWDomain.WEIGHT_TRANSIT_TIME");
    public static final Parameter<Double> WEIGHT_ROUTE_DURATION = new Parameter<>("DARPTWDomain.WEIGHT_ROUTE_DURATION");
    public static final Parameter<Double> WEIGHT_SLACK_TIME = new Parameter<>("DARPTWDomain.WEIGHT_SLACK_TIME");
    public static final Parameter<Double> WEIGHT_RIDE_TIME = new Parameter<>("DARPTWDomain.WEIGHT_RIDE_TIME");
    public static final Parameter<Double> WEIGHT_EXCESS_RIDE_TIME = new Parameter<>("DARPTWDomain.WEIGHT_EXCESS_RIDE_TIME");
    public static final Parameter<Double> WEIGHT_WAIT_TIME = new Parameter<>("DARPTWDomain.WEIGHT_WAIT_TIME");

    @LoadsComponent({ SolutionBuilder.class, SolutionHandler.class })
    public static void loadSolutionData(ComponentRegister cr,
                                        ParameterRegister pr,
                                        ProblemInstance pi)
    {
        MutableFactorMap weightsMap = new MutableFactorMap();
        weightsMap.addFactor(Factor.TRANSIT_TIME, FactorValue.create(pr.getRequiredValue(WEIGHT_TRANSIT_TIME)));
        weightsMap.addFactor(Factor.ROUTE_DURATION, FactorValue.create(pr.getRequiredValue(WEIGHT_ROUTE_DURATION)));
        weightsMap.addFactor(Factor.SLACK_TIME, FactorValue.create(pr.getRequiredValue(WEIGHT_SLACK_TIME)));
        weightsMap.addFactor(Factor.RIDE_TIME, FactorValue.create(pr.getRequiredValue(WEIGHT_RIDE_TIME)));
        weightsMap.addFactor(Factor.EXCESS_RIDE_TIME, FactorValue.create(pr.getRequiredValue(WEIGHT_EXCESS_RIDE_TIME)));
        weightsMap.addFactor(Factor.WAIT_TIME, FactorValue.create(pr.getRequiredValue(WEIGHT_WAIT_TIME)));
        weightsMap.addFactor(Factor.TIME_WINDOWS_VIOLATION, FactorValue.create(pi.getClientsCount()));
        weightsMap.addFactor(Factor.MAXIMUM_RIDE_TIME_VIOLATION, FactorValue.create(pi.getClientsCount()));
        weightsMap.addFactor(Factor.MAXIMUM_ROUTE_DURATION_VIOLATION, FactorValue.create(pi.getClientsCount()));

        SolutionBuilder sb = cr.provide(new SolutionBuilder(pi));
        cr.provide(new SolutionHandler(pi, sb, weightsMap));
    }

    @LoadsComponent(ProblemInstance.class)
    public static void loadProblemInstance(ComponentRegister cr, ParameterRegister pr)
    {
        String file = pr.getRequiredValue(INSTANCE);
        cr.provide(DARPTWFactory.getInstance().createProblemInstance(file));
    }

    @LoadsComponent(DARPTWDomain.class)
    public static void loadDomain(ComponentRegister cr,
                                  ProblemInstance pi,
                                  SolutionHandler sh,
                                  SolutionBuilder sb)
    {
        cr.provide(new DARPTWDomain(sh, pi, sb));
    }

    //...

    private DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
    {
        //...
    }

    //...
}
```

Listing 13: A final version of loader methods for the *DARPTWDomain* class, with dependencies resolved.

```java
@Test
public void moduleTest()
{
    DARPTWDomain domain = new ModuleLoader().
        loadAll(DARPTWDomain.class). // loads all three loader methods
        setParameter(DARPTWDomain.INSTANCE, "benchmark-file").
        setParameter(DARPTWDomain.WEIGHT_TRANSIT_TIME, 8.0).
        setParameter(DARPTWDomain.WEIGHT_RIDE_TIME, 0.0).
        setParameter(DARPTWDomain.WEIGHT_EXCESS_RIDE_TIME, 3.0).
        setParameter(DARPTWDomain.WEIGHT_WAIT_TIME, 1.0).
        setParameter(DARPTWDomain.WEIGHT_SLACK_TIME, 0.0).
        setParameter(DARPTWDomain.WEIGHT_ROUTE_DURATION, 1.0).
        getInstance(DARPTWDomain.class);

    Statement heuristic = block(
        domain.loadNewSolution(), // initHeuristic is included in loadNewSolution()!
        domain.moveRandomClientHeuristic(),
        domain.saveSolution(),
        domain.reportSolution()
    );

    OutputManager.getCurrent().setOutputsFromConfig(new OutputConfig().
        addSystemOutputId(DARPTWOutputIds.RESULT_DETAIL)
    );

    heuristic.run();
}
```

Listing 14: A final version of the test method, using all parameters defined.

2.5 Implementing configurable and selectable operators

Consider now that it is desired to add support for alternative solution initialization procedures. These alternatives could be provided and dynamically configured during module loading. This feature should be added in the *DARPTWDomain* module. The main issue with this request is related with dependencies. Imagine that a new initialization procedure is implemented in module *X*, and it is wanted to provide it as a selectable component for the *DARPTWDomain* module. This component surely will be provided as a *Statement* instance. As parameters are set during configuration, it is no possible to use them for the initialization procedure, because the assembling is performed after configuration. In other terms, no *Statement* instance of the new procedure is available when loading modules with the *ModuleLoader* class. Another approach is to create some component in *DARPTWDomain* that allows modules such as *X* to register their implementations. The problem is that, probably, *X* would require some components from *DARPTWDomain* to implement the procedure, therefore a cyclic dependency will be generated.

hMod provides a concrete support for configurable and selectable operations under the concept of *resolvable entities*. It is “resolvable” because these entities should be obtainable *without* depending on the module instance in which they are defined. Placeholders of such entities can be defined within a module, and when all modules are loaded, resolvable entities are automatically configured through user-defined handlers. To support this, hMod define the *Resolvable* interface, from which “identifiers” of resolvable entities can be created and used to reference selectable entities, such as operators.

The first step to implement selectable initialization procedures, is to define a resolvable interface that can conceptually group all of them. This is simply done by extending the *Resolvable* interface with a new name, such as *DARPTWInitializer*. Listing 15 shows the code of this simple component. *Resolvable* is a generic class, therefore it is necessary that the generic type corresponds to the type to be resolved. In this case, it is desired that initialization procedures would be implemented as *Statement* instances.

```
public interface DARPTWInitializer extends Resolvable<Statement>
{
}
```

Listing 15: An example of a resolvable entity for DARPTW initialization procedures.

The second step is to implement a placeholder for the initialization procedure into the module class, rather than using the random heuristic implemented before. For this, hMod provides a *PlaceholderStatement* type which can be used as a *fake* statement, only for loading purposes, until the real statement is finally resolved. Listing 16 shows a modification of the previous implementation with a *PlaceholderStatement* instance included. Note that it is possible to restrict the type of statement accepted by the placeholder through its generic type.

```
public final class DARPTWDomain
{
    //...
    private final PlaceholderStatement<Statement> init = new PlaceholderStatement<>();
    //...

    public DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
    {
        //...

        init = block(
            sh::loadEmptySolution,
            init
        );

        //...
    }

    //...
}
```

Listing 16: A modification of the previous *DARPTWDomain* implementation, including a placeholder for the initialization procedure

The last step is to modify the main loader method to accept a parameter of the defined resolvable type, and to add a *bound handler* on it. The latter is a callback called when resolving can be done. Through this callback, the *Statement* instance configured as initialization procedure into the placeholder can be manually set, which makes the final linking. The modified loader method and a new parameter to select the procedure are shown in Listing 17. The bounding is concretely performed in line 18 through the *addBoundHandler()* method of the *ParameterRegister* class (note that it should be added to the arguments). The method receives two arguments: a parameter whose generic type must be compatible with *Resolvable*, and the callback to configure the resolvable entity. As the loader method resides in the same *DARPTWDomain* class,

```
public final class DARPTWDomain
{
    //...
    public static final Parameter<DARPTWInitializer> INITIALIZER =
        new Parameter<>("DARPTWDomain.INITIALIZER");

    //...

    @LoadsComponent(DARPTWDomain.class)
    public static void loadDomain(ComponentRegister cr,
                                  ParameterRegister pr,
                                  ProblemInstance pi,
                                  SolutionHandler sh,
                                  SolutionBuilder sb)
    {
        DARPTWDomain domain = cr.provide(new DARPTWDomain(sh, pi, sb));
        DARPTWInitializer initResolver = pr.getRequiredValue(INITIALIZER);
        pr.addBoundHandler(initResolver, (st) -> domain.init.set(st));
    }

    //...
}
```

Listing 17: An example for using bound handlers to resolvable entities.

it is possible to access private members of it. Therefore, in the execution code of the callback, it directly sets the `init` placeholder with the value configured for the parameter. Such value is passed to the callback towards the `st` variable in the lambda expression shown in the code.

With the above steps, a configurable parameter for changing dynamically the initialization procedure was defined. Other modules can now define their own procedures under the *DARPTWInitializer* type. To provide a basic alternative, the original random initialization procedure will be provided as a selectable entity. This can be easily done by creating a *bounded resolvable instance*. This is a resolvable instance that can be created with a specific subtype, and whose resolution is directly bounded to a instance of an external entity, such as a module. In this way, it is possible to create a *DARPTWInitializer* instance that, when resolved, returns the instance of the random initializer stored in the `initHeuristic` attribute of the *DARPTWDomain* class. Listing 18 shows how to create the selectable element within such class. The factory

```
public final class DARPTWDomain
{
    //...

    public static final DARPTWInitializer DEFAULT_INIT = Resolvable.boundTo(
        DARPTWInitializer.class,
        DARPTWDomain.class,
        (domain) -> domain.initHeuristic
    );

    //...
}
```

Listing 18: An example of creating a typed and bounded *Resolvable* instance, which can be used as a selectable entity.

method *boundTo()* of the *Resolvable* class is used. The first argument is the *Resolvable* subtype to be instantiated (an interface must be provided). This allows to “mask” an internal *Resolvable* instance through a proxy instance of the *DARPTWInitializer* type. The second argument is the class from which the resolvable instance will be obtained, in this case, the same *DARPTWDomain* class. The third argument is a callback called when the entity must be resolved. In the code, a lambda expression is used, where the `domain` argument type corresponds to the bounded type, i.e., *DARPTWDomain*, and the callback return type must correspond to the type of the solved entity, i.e., *Statement*. Again, as the resolvable was defined within the *DARPTWDomain* class, it can directly access to its private `initHeuristic` attribute.

With the above selectable item, it is possible to modify the test method to support the new features added. The modified code is shown in Listing 19. Note that the only change done

```
@Test
public void moduleTest()
{
    DARPTWDomain domain = new ModuleLoader().
        loadAll(DARPTWDomain.class).
        //...
        setParameter(DARPTWDomain.INITIALIZER, DARPTWDomain.DEFAULT_INIT).
        getInstance(DARPTWDomain.class);

    //...
}
```

Listing 19: A modified test method for supporting the selectable initializer procedure.

was the adding of the new parameter and its value, which corresponds to the random initializer implementation. In this way, the *ModuleLoader* component will trigger the callback of the `DARPTWDomain.INITIALIZER` resolvable, passing as argument the resolved *Statement* instance obtained from the `DARPTWDomain.DEFAULT_INIT` callback.

Another use case for resolvable entities is to create a set of selectable operators under a common type. For example, suppose that it is desired to create a generic selectable type for DARPTW-related heuristics, with the possibility that any module can provide a typed subset of them. For this, a new type named *DARPTWHeuristic* can be created to extend *Resolvable*, just like the one in Listing 15. Additionally, in the *DARPTWDomain* module, a set of default heuristics would be provided, in which the *move random client* heuristic could be included. hMod provides a simple structure with which the mentioned default heuristics could be grouped, rather than define a single attribute fore each one: the *Selector* class. Additionally, there is a special resolvable type for create a subset of *DARPTWHeuristic*-like types: the *SelectableValue* class, which can be used alongwith a *Selector* instance. The main idea is that each module that wants to define its own *DARPTWHeuristic* subset should implement a custom type that implement such interface and extend *SelectableValue* at the same time.

Listing 20 shows an example of *Selector* and *SelectableValue* applied to the *DARPTWDomain* module. The following elements were added or changed to the class:

- The inner class *DefaultHeuristic* represents the subset of *DARPTWHeuristic* exclusive for the *DARPTWDomain* module. Note that its constructor is private, therefore no other modules different than *DARPTWHeuristic* can create new instances of the subset. When

```
 1  public final class DARPTWDomain
 2  {
 3      public static class DefaultHeuristic extends SelectableValue<Statement> implements DARPTWHeuristic
 4      {
 5          private DefaultHeuristic()
 6          {
 7              super(DARPTWDomain.class, (d) -> d.heuristics);
 8          }
 9      }
10
11      //...
12
13      public static final DefaultHeuristic MOVE_RANDOM_CLIENT = new DefaultHeuristic();
14
15      //...
16      private final Selector<DefaultHeuristic, Statement> heuristics = new Selector<>(); // replaces '
           moveRandomClientHeuristic'
17
18      private DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
19      {
20          //...
21
22          heuristics.add(MOVE_RANDOM_CLIENT, block(() -> {
23              // same code of move random client
24          }));
25
26          //...
27      }
28
29     //...
30      public Statement heuristic(DefaultHeuristic h) { return heuristics.get(h); } //replaces 'moveRandomClientHeuristic'
           getter
31  }
```

Listing 20: An example for the use of the *Selector* alongwith *SelectableValue*.

extending *SelectableValue*, two arguments must be provided to the parent constructor: the container class, i.e., the class from which instances of the implemented type will be obtained, and a callback for obtaining such instances. This callback takes as argument an instance of the container class, i.e., the *DARPTWDomain* class, and must return an instance of *Selector*.

- The lambda in line 7 is returning the *Selector* instance defined down in line 16. A *Selector* is a generic class for which the first type corresponds to the resolvable subtype associated with it, and the second type is the one that is mapped by instances of the first. In the example, *DefaultHeuristic* are mapped to *Statement*.
- When the class is initialized, the original assignment to `moveRandomClientHeuristic` was replaced in line 22 by adding a new entry to the *Selector* instance. This instance, named `MOVE_RANDOM_CLIENT`, is declared in line 13 as a constant, which is the one finally associated with the *move random client* heuristic.
- The getter in line 30 replaces the original getter of the *move random client* heuristic to accept a *DefaultHeuristic* instance as argument, and return the mapped *Statement* instance stored in the *Selector* attribute. As instances of *DefaultHeuristic* can be created only in the module, it is pretty type-safe to request heuristics through this method.

The above changes make more easy to add new heuristics to the module. It is only required to create a new instance of the *DefaultHeuristic* class, add it as a constant in the module class such as `MOVE_RANDOM_CLIENT`, and then add the corresponding mapping to a *Statement* in the constructor code such as line 22. To use this mapping implemented in the *DARPTWDomain* module, the test code could be modified as Listing 21 shows. The only change is that, rather than using the previous getter of the *move random client* heuristic, now the constant mapped to such

```java
@Test
public void moduleTest()
{
    DARPTWDomain domain = new ModuleLoader().
        loadAll(DARPTWDomain.class).
        //...
        getInstance(DARPTWDomain.class);

    Statement heuristic = block(
        domain.loadNewSolution(),
        domain.heuristic(DARPTWDomain.MOVE_RANDOM_CLIENT),
        domain.saveSolution(),
        domain.reportSolution()
    );

    //...
}
```

Listing 21: Changes in the test method for supporting selectable heuristics in the *DARPTWDomain* module.

heuristic is used alongwith the *heuristic()* method. An interesting consequence of this approach is that, if an external module exposes a parameter of type *DARPTWHeuristic*, the constants declared in *DARPTWDomain* can be used to set it, and further resolving will be automatically done by the *ModuleLoader* component, therefore, there will be no need of explicit *Statement* references.

2.6 Including extensible routines

Sometimes it is desirable to provide some "extension points" to specific parts of the algorithm structure. This should allow to add new code within those points and maintain a consistency in the general structure at the same time. For example, consider that it is desired to add a checking routine after initializing a solution for the DARPTW domain. Such routine can be executed just after the configured initializer heuristic. hMod provides the *Routine* interface, which is an extension of *Statement* and adds several methods for adding behavior before and after the main code. The default implementation is the *ValidableRoutine* class, which can be configured for further validation after the module loading process. In this way, it is possible to check if the routine was effectively extended by some module, and if not, exception can be thrown.

Adding routines to modules is a pretty straightforward process. Listing 22 shows how to add the above mentioned checking routine to the *DARPTWDomain* already implemented. The *ValidableRoutine* constructor receives two possible arguments: an identifier for debugging purposes, and a boolean that indicates if the routine must be checked to be empty. By leaving the latter argument or passing `true` to it, if the routine was not extended in any way, an exception will be thrown during the loading process. After defining the routine attribute, it can be appended to the structure section that is wanted to extend. In this case, it is added after the initializer heuristic in line 14. The final method declared in line 24 is a *processor method*. When adding validating routines to a module, it is mandatory to implement one of these methods, which will be called at the end of the loading process. It can have any name, must be a non-args method with no return type, and can have any access modifier. Also, it must be tagged with the *Processable* annotation. Within the method, post-processing code can be added to the routines, but it is mandatory to call

```
public final class DARPTWDomain
{
    //...
    private final ValidableRoutine solutionCheck =
        new ValidableRoutine("DARPTWDomain.solutionCheck", false);

    private DARPTWDomain(SolutionHandler sh, ProblemInstance pi, SolutionBuilder sb)
    {
        //...

        loadNewSolution = block(
            sh::loadEmptySolution,
            init,
            solutionCheck
        );

        //...
    }

    //...
    public Routine solutionCheck() { return solutionCheck; }

    @Processable
    private void process()
    {
        solutionCheck.validate();
    }
}
```

Listing 22: An example for the use of extensible and validable routines.

the *validate()* method at some point for each routine declared in the class. This will finally build the routine alongwith all its extensions.

To check the usage of the newly created routine, a dummy module that depends on the *DARPTWDomain* one can be implemented. Listing 23 shows the code of such module. Different append and prepend methods can be used to extend the original structure. It is important to remark that extension of a single routine across several modules is performed according to dependencies and load ordering. That is: if a module *A* requires another *B* to be loaded, *A* will be always loaded before *B*. If *A* and *B* has no interdependencies, they will be loaded in order of appearance when *ModuleLoader* is used, only if *A* and *B* are not required by other modules to be loaded. Therefore, this ordering will determine the extension order of a single routine in *A* and *B* as well.

Finally, the test method can be updated to include the use of the routine in the dummy module. Listing 24 shows this usage. Note that now, the *loadAll()* method includes the new *DummyDARPTW* class for loading. This will enable the extension points for the routine defined in this example. Additionally, note the code at line 9. As the construction of extended routines is done during loading, no further statements can be appended after calling the *getInstance()* method. Therefore, such code will not have any effect on the built algorithm structure.

2.7 Module integration

With the *DARPTWDomain* class, there is a packaged module ready to use in different contexts. Several procedures are provided by this module, and it can be further extended to add new ele-

```java
public class DummyDARPTW
{
    private static Statement createMessage(String msg)
    {
        return () -> OutputManager.println(DARPTWOutputIds.RESULT_DETAIL, msg);
    }

    @LoadsComponent
    public static void load(DARPTWDomain domain)
    {
        domain.solutionCheck().
            apply((st) -> block(createMessage("This is the main code!"), st)).
            append(createMessage("Just after the main code.")).
            appendBefore(createMessage("At the end of the code before the main code.")).
            appendAfter(createMessage("At the end of the code after the main code.")).
            prepend(createMessage("Just before the main code.")).
            prependBefore(createMessage("At the start of the code before the main code."
                )).
            prependAfter(createMessage("At the start of the code after the main code."))
                ;
    }
}
```

Listing 23: Example of a module that uses a routine defined of in module.

```java
@Test
public void moduleTest()
{
    DARPTWDomain domain = new ModuleLoader().
        loadAll(DARPTWDomain.class, DummyDARPTW.class).
        //...
        getInstance(AltDARPTWDomain.class);

    domain.solutionCheck().append(
        () -> OutputManager.println("This won’t have absolutely no effect!")
    );

    //...
}
```

Listing 24: A modified test method for supporting the selectable initializer procedure.

ments. However, for supporting complex algorithmic architectures, features offered by modules like *DARPTWDomain* would be integrated with different features of other modules. Suppose that it is desired now to construct some kind of iterative algorithm using the simpler heuristics implemented in the DARPTW module. Rather than changing directly the module already implemented in the *DARPTWDomain* class, it is possible to use other ready-to-use modules that provide the new features requested and, through an integrative approach, a new bigger module could be constructed. In practice, modules can only provide new features, extend features of already built modules, or a mix of both. The latter is the most common case.

In this example, a new module will be implemented based on *DARPTWDomain* and the *common heuristic* module already implemented in hMod. The frontend class of this module, the *IterativeHeuristic* class, allows to create and configure iterative algorithms in a straightforward manner. These algorithms can be configured to terminate at a specific number of iterations or time elapsed. Initialization and finalization routines can be configured as well. Listing 25 shows the integrated module code, which was named *IterativeDARPTW*. The loader method exposes

```java
public final class IterativeDARPTW
{
    private static Statement print(Supplier<String> sup)
    {
        return () -> OutputManager.println(HeuristicOutputIds.EXECUTION_INFO, sup.get())
            ;
    }

    @LoadsComponent
    public static void load(IterativeHeuristic iHeu,
                            IterationHandler ih,
                            TimeElapsedHandler teh,
                            DARPTWDomain darptw)
    {
        iHeu.initReporting().append(print(() -> "This is printed at the start."));

        iHeu.finishReporting().append(block(
            print(() -> "This is printed at the end."),
            darptw.reportSolution()
        ));

        iHeu.init().append(darptw.loadNewSolution());
        iHeu.finish().append(darptw.saveSolution());

        iHeu.iteration().append(block(
            print(() -> "This is the iteration number " + ih.getCurrentIteration()),
            print(() -> "There are " + teh.getElapsedSeconds() + " secs. elapsed"),
            darptw.heuristic(DARPTWDomain.MOVE_RANDOM_CLIENT)
        ));
    }

    private IterativeDARPTW()
    {
    }
}
```

Listing 25: An example of how module features can be integrated into a new module.

the dependencies mentioned before, i.e., a *IterativeHeuristic* and *DARPTWDomain* instances. Additionally, *IterationHandler* and *TimeElapsedHandler* are requested for printing information regarding iterations and time elapsed. *IterativeHeuristic* exposes several routines useful for the configuration of an iterative algorithm. The *initReporting()* and *finishReporting()* routines are suited for printing initialization and finalization messages, respectively. The *init()* and *finish()* routines are called at the start and end of the algorithm, respectively. Finally, the *iteration()* routine contains the main iteration block. It is important to remark that iteration data is automatically updated by the *IterativeHeuristic* module, the user do not need to advance iterations manually. Concretely, the new module implements a simple iterative algorithm that performs a number of iterations and, on each iteration, print some messages and executes the *move random client* heuristic implemented before.

Note that no new instances were implemented by the *IterativeDARPTW* module. All configuration was put in the loader method. There are situations in which it is desired to provide new routines accessible from the module instance. This could make the module further extensible. Listing 26 shows a simple refactoring of the previous module code to support an extensible point of the iteration block through the `main` attribute.

The final step in the integration of modules is to load them with the *ModuleLoader* component. In this case, a new test method will be implemented. Listing 27 shows the code of it. Note that the *loadAll()* method includes all the involved modules. Parameters of *DARPTWDomain* were included such as previous examples, and additionally, parameters of the *IterativeHeuristic* module are required. `IterativeHeuristic.MAX_ITERATIONS` configures the maximum number of iterations, and `IterativeHeuristic.MAX_SECONDS` the maximum algorithm duration in seconds. Additionally, a new output was configured: `HeuristicOutputIds.EXECUTION_INFO`. To grab the algorithm instance, the *Heuristic* interface through the *getInstance()* method must be requested, that is not more than a specific *Statement* instance.

2.8 Scripting the execution

The modules implemented at this point are enough complete for being distributed and/or being put in a (pre)production environment. Currently, they have been executed only at a testing context. Even for further modifications, it would be adequate to deploy them on a more definitive environment. There is a particular issue in their current state that make this necessary. Through the *ModuleLoader* component, several parameters are configured prior to loading and running. The problem is that values of such parameters are specified in the code of test classes, which cannot be modified without recompiling the related sources. Although components like *jUnit* are enough flexible to create good test suites, in the context of algorithm design, there should be more adequate development alternatives to play with configuration outside Java code.

hMod provides several tools for make the testing and fixing/updating cycle more friendly. The main frontend of such tools is the *Launcher* application. It is a bundled executable class that allows to start a command system. This system can be used directly through console, or scripts files can be implemented to manage it. To run the *Launcher* application, the following steps should be performed:

- Select a folder that will contain the hMod working files. This will be the *executing* folder.
- Put the hMod binaries jar and its dependencies into the executing folder.
- Create two subfolders into the executing folder named `bundles` and `scripts`.
- Run hMod binaries from the executing folder by using the command `java -jar hmod-bin.jar hmod.launcher.Launcher`.

If all goes well, two things would happen. First, a console application would start and showing the text in Listing 28. Second, a file named `hmod.properties` would be created in the executing folder. This is a configuration file in which several *Launcher* settings can be defined. Note that there is an entry like `hmod.launcher.bundlesPaths = bundles`. This configures the path in which module bundles will be searched, and because that the `bundles` subfolder was previously created.

From the console, several commands can be executed. The `help` command provides detailed information about them. In this section, only commands relevant for execution will be generally described. Before this, DARPTW related modules must be packed into a *bundle jar*. This is a Java binary file that contains all compiled sources of a module. They can be dynamically loaded and unloaded by the *BundleLibrary* component. For allowing this, *it is mandatory*

```java
public final class IterativeDARPTW
{
    private static Statement print(Supplier<String> sup)
    {
        return () -> OutputManager.println(HeuristicOutputIds.EXECUTION_INFO, sup.get())
            ;
    }

    @LoadsComponent(IterativeDARPTW.class)
    public static void load(ComponentRegister cr,
                            IterativeHeuristic iHeu,
                            IterationHandler ih,
                            TimeElapsedHandler teh,
                            DARPTWDomain darptw)
    {
        cr.provide(new IterativeDARPTW(iHeu, ih, teh, darptw));
    }

    private final ValidableRoutine main = new ValidableRoutine("IterativeDARPTW.main");

    private IterativeDARPTW(IterativeHeuristic iHeu,
                            IterationHandler ih,
                            TimeElapsedHandler teh,
                            DARPTWDomain darptw)
    {
        main.append(block(
            print(() -> "This is the iteration number " + ih.getCurrentIteration()),
            print(() -> "There are " + teh.getElapsedSeconds() + " secs. elapsed"),
            darptw.heuristic(DARPTWDomain.MOVE_RANDOM_CLIENT)
        ));

        iHeu.initReporting().append(
            print(() -> "This is printed at the start.")
        );

        iHeu.init().append(darptw.loadNewSolution());
        iHeu.iteration().append(main);
        iHeu.finish().append(darptw.saveSolution());

        iHeu.finishReporting().append(block(
            print(() -> "This is printed at the end."),
            darptw.reportSolution()
        ));
    }

    public Routine main() { return main; }
    @Processable private void process() { main.validate(); }
}
```

Listing 26: A refactored version of *IterativeDARPTW* to support the module as an instance.

```
@Test
public void iterativeTest()
{
    Heuristic heuristic = new ModuleLoader().
        loadAll(
            IterativeHeuristic.class,
            DARPTWDomain.class,
            IterativeDARPTW.class
        ).
        setParameter(AltDARPTWDomain.INSTANCE, "benchmark-file").
        setParameter(AltDARPTWDomain.WEIGHT_TRANSIT_TIME, 8.0).
        setParameter(AltDARPTWDomain.WEIGHT_RIDE_TIME, 0.0).
        setParameter(AltDARPTWDomain.WEIGHT_EXCESS_RIDE_TIME, 3.0).
        setParameter(AltDARPTWDomain.WEIGHT_WAIT_TIME, 1.0).
        setParameter(AltDARPTWDomain.WEIGHT_SLACK_TIME, 0.0).
        setParameter(AltDARPTWDomain.WEIGHT_ROUTE_DURATION, 1.0).
        setParameter(AltDARPTWDomain.INITIALIZER, DARPTWDomain.DEFAULT_INIT).
        setParameter(IterativeHeuristic.MAX_ITERATIONS, 100).
        setParameter(IterativeHeuristic.MAX_SECONDS, 30.0).
        getInstance(Heuristic.class);

    OutputManager.getCurrent().setOutputsFromConfig(new OutputConfig().
        addSystemOutputId(DARPTWOutputIds.RESULT_DETAIL).
        addSystemOutputId(HeuristicOutputIds.EXECUTION_INFO)
    );

    heuristic.run();
}
```

Listing 27: A test method that uses integrated modules.

```
Initializing hMod launcher components...
Launching console...

******************************
**** hMod launcher console ***
******************************

Type 'help' for view available commands

> _
```

Listing 28: hMod *Launcher* console presentation.

that bundle jars do not be included into the Launcher classpath. Doing this will load the bundles alongwith the application class loader and no further reloading would be possible. Suppose that modules implemented in previous sections are packed into a `hmod-domains-darptw.jar` file. For allowing them to be recognized as bundles, the `META-INF/MANIFEST.MF` file must be modified to include the entry shown in Listing 29. The relevant entry is `Name`, which must point to the package in which modules are defined. With this modification included, the `hmod-domains-darptw.jar` file must be put into the `bundles` folder, and the `reloadBundles` command should be run into the *Launcher* console. This will reload the modules and will allow to run the updated algorithms on them. If further modifications are required to the components of the `hmod-domains-darptw.jar` bundle, the `reloadBundles` command can be executed many times as needed.

```
Name: hmod/domains/darptw/
Implementation-Vendor: optefx
Implementation-Title: hMod Dial-a-Ride Problem with Time Windows (DARPTW)
Implementation-Version: 1.0.0
```

Listing 29: Example of an entry in `META-INF/MANIFEST.MF` file required to allow a module bundle jar to be recognized by the *Launcher* component.

Now, it is possible to create a script to dynamically run the module tests implemented before. The *Launcher* component uses *javascript* code for implement scripts. It is supported by the default Nashorn engine implemented in Java 8. The main advantage of this is that anything that can be written in javascript code can be applied to the execution of algorithms in hMod. With javascript, it is possible to build complex experiments that include controlled repetitions, parametrized functions, object-like entities for experiment packaging, among others. By default, script files (i.e., js files) can be put into the `scripts` folder, which is already configured in `hmod.properties`. This will help to reduce the writing of commands for calling them. Consider a script named `darptw-test.js` in the `scripts` folder with the content specified in Listing 30. There are many similarities in this code regarding the one in Listing 27. This is because the common points in the Java and javascript syntax. At the start, the `JavaImporter` object is used to include the classes of the required packages. This is some kind of `import` clause provided by Nashorn. From line 26, there are some calls that are particular for script files. The `console` object is declared as a global variable, and allows to access *Launcher* commands. With it, all commands that are runnable from the console can be executed as a function of this object. The function names are the command keywords, and arguments of the commands should be passed as arguments of called functions. In the example code, the following commands are used (others than `reloadBundles`, that is already explained):

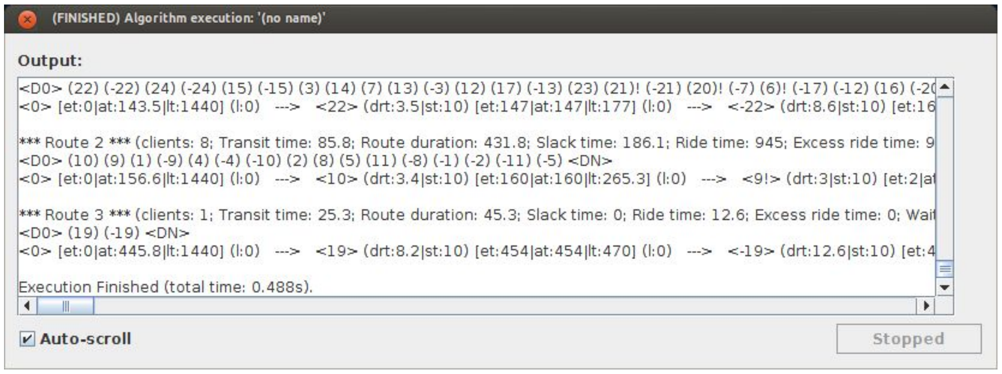
- `threading`: Enables o disables the use of threads for running algorithms. If enabled, the console will continue its execution immediately after starting any algorithm instance execution.
- `setInterface`: Configures the interface to show when executing algorithms instances. There are two options: `default` and `windowed`. The former shows system output in console, while the latter shows system output in a window.
- `debug`: Enables o disables debugging. If enabled, full details of exceptions will be thrown to system output.
- `randomSeed`: Sets the random seed to use in the next algorithm instance execution.
- `clearOutputs`: Clears all current output configuration added through commands.
- `addOutput`: A console wrapper of the *OutputManager* component. Enables system output towards a specific output identifier.
- `echo`: Throws output to console.
- `run`: Starts the execution of the provided *Statement* instance. This will handle different elements related to execution, such as random seed initialization, interface configuration, output configuration, threading support, among others. This is the correct way to run algorithms through console.

```
var includes = new JavaImporter(
    Packages.hmod.solvers.common,
    Packages.hmod.domains.darptw,
    Packages.optefx.loader
);

with(includes) {
    var heuristic = new ModuleLoader().
        loadAll(
            IterativeHeuristic.class,
            DARPTWDomain.class,
            IterativeDARPTW.class
        ).
        setParameter(DARPTWDomain.INSTANCE, "benchmark-file").
        setParameter(DARPTWDomain.WEIGHT_TRANSIT_TIME, 8.0).
        setParameter(DARPTWDomain.WEIGHT_RIDE_TIME, 0.0).
        setParameter(DARPTWDomain.WEIGHT_EXCESS_RIDE_TIME, 3.0).
        setParameter(DARPTWDomain.WEIGHT_WAIT_TIME, 1.0).
        setParameter(DARPTWDomain.WEIGHT_SLACK_TIME, 0.0).
        setParameter(DARPTWDomain.WEIGHT_ROUTE_DURATION, 1.0).
        setParameter(DARPTWDomain.INITIALIZER, DARPTWDomain.DEFAULT_INIT).
        setParameter(IterativeHeuristic.MAX_ITERATIONS, 100).
        setParameter(IterativeHeuristic.MAX_SECONDS, 30.0).
        getInstance(Heuristic.class);

    console.reloadBundles();
    console.threading(false);
    console.setInterface("windowed");
    console.debug(true);
    console.randomSeed(1234);

    console.clearOutputs();
    console.addOutput(DARPTWOutputIds.RESULT_DETAIL);
    console.addOutput(HeuristicOutputIds.EXECUTION_INFO);

    console.echo("Starting test execution!");
    console.run(heuristic);
}
```

Listing 30: An example of a javascript file runnable in the *Launcher* environment.

To run the script, the `runScript` command must be called in the console. The name of the script path must be passed as argument, starting from the `script` folder. In this case, the call is `runScript darptw-test.js`. For the above script file, this execution should trigger the showing of a window with the algorithm execution, such as Figure 4 illustrates. This is because `windowed` interface was selected in the script.

With the current implementation, now it is possible to change the parameters of the algorithm without recompiling or making changes to Java files, only the script must be changed. Additionally, it is possible to achieve the same script execution from the command line, without entering the *Launcher* console. For this, the application must be started as `java -jar hmod-bin.jar hmod.launcher.Launcher darptw-test.js`. This will immediately start the script after loading *Launcher* components, and after finish, the application will close automatically.



```
<D0> (22) (-22) (24) (-24) (15) (-15) (3) (14) (7) (13) (-3) (12) (17) (-13) (23) (21)! (-21) (20)! (-7) (6)! (-17) (-12) (16) (-2
<0> [et:0|at:143.5|lt:1440] (l:0)   --->   <22> (drt:3.5|st:10) [et:147|at:147|lt:177] (l:0)   --->   <-22> (drt:8.6|st:10) [et:16

*** Route 2 *** (clients: 8; Transit time: 85.8; Route duration: 431.8; Slack time: 186.1; Ride time: 945; Excess ride time: 9
<D0> (10) (9) (1) (-9) (4) (-4) (-10) (2) (8) (5) (11) (-8) (-1) (-2) (-11) (-5) <DN>
<0> [et:0|at:156.6|lt:1440] (l:0)   --->   <10> (drt:3.4|st:10) [et:160|at:160|lt:265.3] (l:0)   --->   <9!> (drt:3|st:10) [et:2|at

*** Route 3 *** (clients: 1; Transit time: 25.3; Route duration: 45.3; Slack time: 0; Ride time: 12.6; Excess ride time: 0; Wait
<D0> (19) (-19) <DN>
<0> [et:0|at:445.8|lt:1440] (l:0)   --->   <19> (drt:8.2|st:10) [et:454|at:454|lt:470] (l:0)   --->   <-19> (drt:12.6|st:10) [et:4

Execution Finished (total time: 0.488s).
```

Figure 4: A windowed execution of an algorithm using the hMod *Launcher* application.

3 Final comments

hMod implementation flexibility allows to address the construction of algorithm components and further assembling in many different ways. Therefore, it is important to construct additional tools on top of the framework that allows to provide a better guide when addressing particular algorithm scenarios. The *IterativeHeuristic* module is an example of how smaller but useful components should be provided to make common design problems more straightforward to resolve.

The framework is actually in an alpha stage, but it currently has enough components for building complex algorithmic architectures. It is important to remark that the current use cases of the framework are oriented to *manual assembling* of such architectures. However, given the extensibility means provided by the framework, it will be possible to extend their core to support adequate semantics for *automatic assembling* as well.

References

[1] Gerardo Berbeglia, Jean-François Cordeau, Irina Gribkovskaia, and Gilbert Laporte. Static pickup and delivery problems: A classification scheme and survey. *Top*, 15:1–31, 2007.

[2] Jean-François Cordeau and Gilbert Laporte. The dial-a-ride problem (darp): Variants, modeling issues and algorithms. *4OR*, 1:89–101, 2003.

[3] Oracle Corporation. Project nashorn.

[4] R M Jørgensen, J Larsen, and K B Bergvinsdottir. Solving the dial-a-ride problem using genetic algorithms. *The journal of the Operational Research Society*, 58:1321–1331, 2007.

[5] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *ACM SIGPLAN Notices*, 33:36–44, 10 1998.