

# OREX-J: towards a universal software framework for the experimental analysis of optimization algorithms

Jan Christian Lang · Thomas Widjaja

Published online: 21 April 2012  
© Springer-Verlag 2012

**Abstract** The Operations Research EXperiment Framework for Java (OREX-J) is an object-oriented software framework that helps users to design, implement and conduct computational experiments for the analysis of optimization algorithms. As it was designed in a generic way using object-oriented programming and design patterns, it is not limited to a specific class of optimization problems and algorithms. The purpose of the framework is to reduce the amount of manual labor required for conducting and evaluating computational experiments: OREX-J provides a generic, extensible data model for storing detailed data on an experimental design and its results. Those data can include algorithm parameters, test instance generator settings, the instances themselves, run-times, algorithm logs, solution properties, etc. All data are automatically saved in a relational database (MySQL, <http://www.mysql.com/>) by means of the object-relational mapping library Hibernate (<http://www.hibernate.org/>). This simplifies the task of analyzing computational results, as even complex analyses can be performed using comparatively simple Structured Query Language (SQL) queries. Also, OREX-J simplifies the comparison of algorithms developed by different researchers: Instead of integrating other researchers' algorithms into proprietary test beds, researchers could use OREX-J as a common experiment framework. This paper describes the architecture and features of OREX-J and exemplifies its usage in a case study. OREX-J has already been used for experiments in three different areas:

---

J. C. Lang (✉)

Chair of Operations Research, Department of Law, Business and Economics,  
Technische Universität Darmstadt, Hochschulstraße 1, 64289 Darmstadt, Germany  
e-mail: j.christian.lang@gmail.com

T. Widjaja

Chair of Information Systems, Department of Law, Business and Economics,  
Technische Universität Darmstadt, Hochschulstraße 1, 64289 Darmstadt, Germany  
e-mail: widjaja@is.tu-darmstadt.de

Algorithms and reformulations for mixed-integer programming models for dynamic lot-sizing with substitutions, a simulation-based optimization approach for a stochastic multi-location inventory control model, and an optimization model for software supplier selection and product portfolio planning.

**Keywords** Operations research methodology · Experimental analysis of algorithms · Test bed · Software framework · Object-oriented design · Design patterns

## 1 Introduction

Many OR scientists spend a considerable amount of time with the implementation, execution and analysis of computational experiments to evaluate optimization algorithms. This paper presents a software framework<sup>1</sup>—the Operations Research EXperiment Framework for Java (OREX-J)—that aims at supporting those activities by automating them as much as possible and providing a generic “test bed”<sup>2</sup> architecture.

Figure 1 shows an idealized model of the process flow during the development of an algorithm for a real-world OR optimization problem. The activities (esp. those related to the usage of a test bed) that are facilitated by OREX-J are highlighted by a gray box.

Typically, a project begins with a requirements analysis to determine the assumptions of the optimization problem that are used to formulate a mathematical model. After that or in parallel, data of real-world problem instances need to be gathered, or, if this does not seem viable, assumptions for a test instance generator need to be developed. The next step is to select one or more potentially suitable algorithms or model reformulation approaches for the problem. After that, the scientist herself or a software developer implements prototypes of the algorithms. In order to compare her algorithms with rival algorithms, the scientist then conducts a computational experiment<sup>3</sup> by running the algorithms on a set of real-world or generated test instances and analyzing the results and algorithm behavior, e.g. in terms of solution quality and run-times.

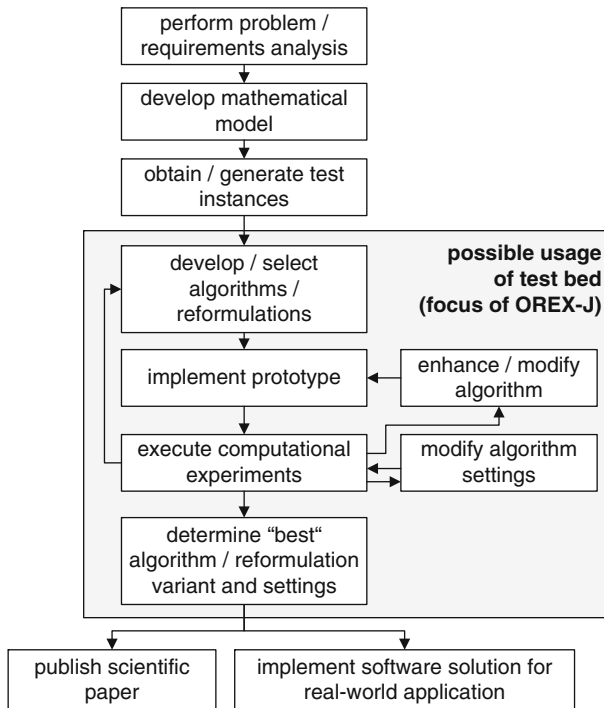
De facto, the development of an algorithm for an OR problem often takes place as an iterative process in a trial-and-error manner involving numerous algorithm modifications and parameter fine-tuning. If the quality of solutions yielded by the algorithm seems too low or the algorithm run-times excessively high, the following three types of actions might follow:

---

<sup>1</sup> See Sect. 3.1 for a definition of the term “software framework”.

<sup>2</sup> In the following, we use the term “test bed” to refer to a software environment for conducting computational experiments, whereas the term is used synonymously with “set of test instances for an optimization problem” in some publications.

<sup>3</sup> In our terminology, an “*experimental unit*” is a task that solves one problem instance of an optimization problem using one specific algorithm with respective parameter settings and given random seeds. By “experiment”, we refer to a set of experimental units created e.g. by using a factorial design that tries all combinations of values for two or more factors, where the factors represent various algorithm or problem instance generator settings.



**Fig. 1** Idealized OR solution development process

1. “Modify algorithm settings”: If the scientist assumes that the unwanted behavior of the algorithm is caused by suboptimal parameter settings, she changes those settings (e.g. by varying the tabu list length of a tabu search metaheuristic) or adapts a procedure that determines the settings automatically.
2. “enhance/modify algorithm”: The scientist modifies or enhances the algorithm in some way to improve its effectiveness or efficiency (e.g. by using a different cross-over operator for a genetic algorithm or adding a new type of valid inequalities in a branch&cut algorithm).
3. “develop/select algorithm/reformulation”: the scientist selects a different algorithm type (or reformulation) that seems promising for the considered optimization problem and proceeds with testing this algorithm.

After each of these actions, the computational experiments as well as successive analysis such as aggregations of results and statistical tests have to be rerun, which may require significant manual work.

If an algorithm/reformulation with respective parameter settings has been found that is deemed sufficiently good, two steps can follow: First, the approach is usually documented in a scientific publication. Second, the algorithm is implemented as a software for production usage, either based on the prototype source code, or by re-implementing the algorithm from scratch. For the sake of usability, the latter step might include developing a decision support system (DSS) for visualizing, analyzing, and manipulating problem instances and solutions.

	alternative 1: “quick-and-dirty” approach	alternative 2: application-specific data model	alternative 3: usage of framework (e.g. OREX-J)
degree of automation of experiment execution	manual / partially automated	partially / fully automated	fully automated
centralization of data storage	decentralized storage in files, partially manual labor	centralized, application- specific storage in relational database	centralized, largely generic storage in relational database
expenditure of time for implementation	+	-	O
expenditure of time for evaluation of experiments and convenience of evaluation of experiments	-	O	++

**Fig. 2** Alternatives for implementing experiment test beds

Figure 2 portrays three fundamental alternatives regarding the implementation of a test bed for computational experiments, ordered by the degree of automation of experiment execution. A common choice is alternative 1 that we term the “*quick-and-dirty*” approach, meaning that a proprietary test environment is developed for a specific algorithm and problem type. This environment usually requires a substantial amount of manual work for executing and analyzing experiments, and uses simple approaches for storing results such as separate log files generated by each algorithm run. Alternative 2 refers to an approach that automatizes the execution of experiments and saves all output data of computational results centrally in a relational database that has a proprietary, application-specific structure. Alternative 3 refers to the usage of a software framework like OREX-J for automatizing computational experiments that offers a largely generic functionality for storing experiment descriptions and output data in a database for later analysis.<sup>4</sup>

The topic of software libraries and code reuse in operations research has already been considered in various publications, see e.g. Fink et al. (1998) and the comprehensive collection edited by Voß and Woodruff (2002). OREX-J is a so-called *software framework*. A software framework is a special type of *software library*<sup>5</sup> with the following distinguishing properties:

- It provides generic functionality that can be specialized by the user of the framework by injecting source code using a predefined mechanism. This can e.g. be performed by providing callback functions in imperative programming languages or implementing interfaces or *inheriting* from abstract classes and *overriding* methods in *object-oriented programming* (OOP).

<sup>4</sup> Figure 2 does not include the time needed for developing frameworks like OREX-J themselves in the implementation effort estimate of alternative 3.

<sup>5</sup> We define the term “software library” as a reusable collection of source code with functions/classes providing a certain functionality, implemented in a programming language.

- The control flow is *inverted*, compared to a typical software library: The source code of the framework calls source code provided by the user, whereas source code of a “normal” software library is called by user source code.
- It predefines the architecture of the software system to a great extent, which potentially reduces the users’ flexibility regarding architectural choices, but saves the effort of designing an sophisticated architecture.
- The framework has a default behavior that can be varied by the user by *injecting* source code at certain spots, but some parts of its structure and behavior cannot be changed.
- Its source code is not meant to be modified by users.

If computational experiments are conducted without using a software framework that automatizes some of the required tasks (especially in the “quick-and-dirty” approach), several difficulties may arise, which we illustrate by the following exemplary situations:

- Executing an experimental design generates dozens of output files (e.g. in comma-separated values (CSV) or proprietary format), separate for each experimental unit, that have to be aggregated and imported into a spreadsheet software manually in a tedious, time-consuming process.
- After a small change made in an algorithm or problem instance generator, a large amount of manual work is required to rerun an experiment and the consequent analyses.
- When resuming the development and analysis of an algorithm implemented some time ago (e.g. for running additional experiments for a revision of a paper), it might be difficult to reproduce computational results if experiments have not been documented painstakingly.
- After running an experiment with several algorithms on a set of large, difficult problem instances for days or weeks, the scientist notices that one piece of information that would be very helpful for the experimental analysis has not been collected and saved in the experiment output files.

The main purpose of OREX-J is to *reduce the amount of manual labor* required for conducting experiments. Several software libraries for implementing optimization algorithms already exist (see Sect. 3.1), but those are usually limited to a specific class of problems and/or algorithms, e.g. branch&cut algorithms for mixed-integer linear programming (MILP) or specific metaheuristics like genetic algorithms. There was no generic software framework that offered comprehensive functionality for executing experiments automatically (batch experiments) and storing their output data in a way that they can easily be analyzed and interpreted.

Based on this motivation we derived the following requirements: The framework should *support batch experiments for experimental designs* ( $REQ_A$ ), be usable for *different problem types* ( $REQ_B$ ) and allow the application of *exact and heuristic algorithms* ( $REQ_C$ ). We wanted to develop a framework that *automates the collection of (centrally available) data* on the experiments ( $REQ_D$ ). Furthermore we aimed to foster the distribution and further development of the framework by providing a *free* ( $REQ_E$ ) and *open source solution* ( $REQ_F$ ) that is based on a *common programming*

language ( $REQ_G$ ). Hence, we only used software components that allow an open source distribution.

OREX-J is not limited to a specific class of optimization problems and algorithms as it was designed in a generic way by utilizing established *software engineering* methodology, in particular *object-oriented programming* and *design patterns* (Gamma et al. 1994). OREX-J allows batch experiments and provides a generic, extensible data model for storing detailed data on an experiment and its results. Those data can include algorithm settings, test instance generator settings, the problem instances themselves, run-times, algorithm logs, solution properties, etc. All data are automatically saved in the *relational database management system (RDBMS)* MySQL<sup>6</sup> by means of the *object-relational mapping (ORM)* library Hibernate.<sup>7</sup> This *persistence*<sup>8</sup> infrastructure simplifies the task of analyzing computational results, as even complex analyses can be performed using comparatively simple Structured Query Language (SQL) queries. It maps each Java class and its subclasses to one database table. Each column in the table corresponds with one class attribute. Each row in the table corresponds with one object. Certain columns that represent generic attributes are pre-defined, whereas application-specific attributes are added as new columns in the corresponding database table. Thus, when using OREX-J for an application, some columns of the tables change, but it is not mandatory to add new tables. The ORM Hibernate automatically generates database tables for Java classes and writes Java objects and their attributes into those tables.<sup>9</sup> This also applies to application-specific data structures.

Besides the reduction of manual work, the usage of OREX-J induces further advantages: By applying OREX-J, the user en passant utilizes *established paradigms* known from literature on experimental algorithmics (such as the usage of variance reduction techniques, and mechanisms to improve reproducibility of experiments and comparability of algorithms Sect. 3.2) and software patterns (see Sect. 3.3). Furthermore, OREX-J simplifies the comparison of algorithms developed by different research groups: Instead of integrating other researchers' algorithms into proprietary, self-developed test beds, OR scientists could reuse it as a common framework for their experiments. In addition, OREX-J could serve as a reference architecture for future frameworks.

OREX-J has already been used for experiments in research projects in three different areas (see Sect. 4.5), proving its applicability and versatility. It is available as free and *open source software* under the Apache License 2.0 via SourceForge.<sup>10</sup>

<sup>6</sup> <http://www.mysql.com>.

<sup>7</sup> <http://www.hibernate.org>.

<sup>8</sup> In this context, "persistence" means that data on an experiment and its results are stored permanently.

<sup>9</sup> This mechanism resembles the model-driven architecture (MDA) approach, which is based on the idea of automatically generating a software implementation from a formal (e.g. graphical) model description (Kleppe et al. 2003).

<sup>10</sup> <http://orex-j.sourceforge.net>.

## 2 Outline

The remainder of this paper is structured as follows: We provide a brief overview of related research in Sect. 3. Section 4 describes the architecture of OREX-J including its underlying *generic data model* and *Common Random Numbers (CRN)* mechanism for ensuring the reproducibility of randomness in computational experiments, and delineates three research projects in which OREX-J was used. Section 5 contains a case study that illustrates the usage of OREX-J for computational experiments that compare algorithms and reformulations for dynamic lot-sizing problems with substitutions (Lang and Shen 2011; Lang and Domschke 2010).

## 3 Related work

### 3.1 OR software frameworks

A number of software frameworks providing functionality for the implementation of OR optimization algorithms and, to a minor extent, for the execution and analysis of computational experiments already exist. Table 1<sup>11</sup> contains a list and classification of such frameworks. We classify each framework by several criteria (also see the requirements defined in Sect. 1):

- “Experimental designs” ( $REQ_{A1}$ ): Does it include support for generating experimental designs?
- “Batch experiments” ( $REQ_{A2}$ ): Does it include functionality for executing experiments automatically, i.e. sequences of experimental units?
- “Problem type” ( $REQ_B$ ): Which classes of optimization problems can it deal with?
- “Exact or heuristic” ( $REQ_{C1}$ ): Does it support exact or heuristic algorithms, or both?
- “Algorithm type” ( $REQ_{C2}$ ): Which types of algorithms can be implemented with the framework?<sup>12</sup>
- “Automatic, centralized persistence” ( $REQ_D$ ): Are descriptions of experiments and their results saved automatically in a central storage so they can be retrieved and analyzed later?
- “Free product” ( $REQ_E$ ): Is it a free (non-commercial) product?
- “Open source” ( $REQ_F$ ): Is its source code available?
- “Programming language/technology” ( $REQ_G$ ): Which user programming languages does it support?

We included selected commercial and non-commercial MILP solvers such as CPLEX, Xpress, and BCP in the list. They can be seen as software frameworks because they are customizable by callback functions and similar mechanisms (for

<sup>11</sup> The abbreviations B&C, B&P&C and MIQP stand for Branch&Cut, Branch&Cut&Price, and mixed-integer quadratic programming, respectively.

<sup>12</sup> Various specialized frameworks providing pre-implemented algorithms exist, see e.g. Jünger and Thiele (2000); do Carmo et al. (2008); Fink and Voß (2002); Gagné and Parizeau (2006); Cahon et al. (2004); Ralphs and Güzelsoy (2006); Di Gaspero and Schaerf (2003); Durillo et al. (2010).

**Table 1** Comparison of software frameworks for OR algorithm implementations and computational experiments

name	experimental designs ( $RQ_{A1}$ )	batch experiments ( $RQ_{A2}$ )	problem type ( $RQ_B$ )	exact or heuristic ( $RQ_{C1}$ )	algorithm type ( $RQ_{C2}$ )	automatic, centralized persistence ( $RQ_D$ )	free product ( $RQ_E$ )	open source ( $RQ_F$ )	programming language/technology ( $RQ_G$ )	scientific literature	URL
<b>OREX-J</b>	✓	✓	generic	both	generic	✓	✓	✓	Java	on hand	<a href="http://orex-j.sourceforge.net">http://orex-j.sourceforge.net</a>
ABACUS	✓	✓	MILP	exact	B&C&P	✓	✓	✓	C++	Jünger and Thienel (2000)	<a href="http://www.informatik.uni-koeln.de/abacus/">http://www.informatik.uni-koeln.de/abacus/</a>
AIMMS/OSI	✓	✓	MILP/MIQP and others	both	various (via OSI)	✓	✓	✓	C++	-	<a href="http://www.aimms.com">http://www.aimms.com</a>
BCP	✓	✓	MILP	exact	B&C&P	✓	✓	✓	C++	-	<a href="https://projects.coin-or.org/Bcp">https://projects.coin-or.org/Bcp</a>
CPLEX/OPL Studio	✓	✓	MILP/MIQP and others	exact	B&C	✓	✓	✓	C++, Java, .NET, Python	-	<a href="http://www.ibm.com/software/integration/optimization/cplex/">http://www.ibm.com/software/integration/optimization/cplex/</a>
EasyAnalyzer	✓	✓	generic	both	local search/meta-heuristics	✓	✓	✓	C++	Di Gaspero et al. (2007)	-
EasyLocal++	✓	✓	generic	heuristic	local search/meta-heuristics	✓	✓	✓	C++	Di Gaspero and Schaerf (2003)	<a href="http://rabu.diagm.unind.it/EasyLocal++/">http://rabu.diagm.unind.it/EasyLocal++/</a>
EasyMeta	✓	✓	generic	heuristic	metaheuristics	✓	✓	✓	Java	do Carmo et al. (2008)	<a href="http://www.easymeta.com/">www.easymeta.com/</a>
GAMS/OSI	✓	✓	MILP/MIQP and others	both	various (via OSI)	✓	✓	✓	C++	-	<a href="http://www.gams.com">http://www.gams.com</a>
HotFrame	✓	✓	generic	heuristic	local search/meta-heuristics	✓	✓	✓	C++	Fink and Voß (2002)	<a href="http://www1.uni-hamburg.de/INT/hotframe/hotframe.html">http://www1.uni-hamburg.de/INT/hotframe/hotframe.html</a>
jMetal	✓	✓	combinatorial multiobjective optimization	heuristic	metaheuristics	✓	✓	✓	Java	Durillo et al. (2010)	<a href="http://jmetal.sourceforge.net">http://jmetal.sourceforge.net</a>
METSib	✓	✓	generic	heuristic	various metaheuristics	✓	✓	✓	C++	-	<a href="http://projects.coin-or.org/metalib">http://projects.coin-or.org/metalib</a>
OpenBEAGLE	✓	✓	generic	heuristic	evolutionary algorithms	✓	✓	✓	C++	Gagné and Parizeau (2006)	<a href="http://beagle.gel.ulaval.ca">http://beagle.gel.ulaval.ca</a>
oMeta	✓	✓	continuous optimization	heuristic	metaheuristics	✓	✓	✓	C++	-	<a href="http://ometa.berlios.de">http://ometa.berlios.de</a>
OpenTS	✓	✓	generic	heuristic	Tabu Search	✓	✓	✓	Java	-	<a href="http://www.coin-or.org/Ots/">http://www.coin-or.org/Ots/</a>
OS: Optimization Services	✓	✓	generic	both	generic	✓	✓	✓	Web Services	Fourer et al. (2008)	<a href="https://projects.coin-or.org/OS">https://projects.coin-or.org/OS</a>
ParadisEO	✓	✓	generic	heuristic	metaheuristics	✓	✓	✓	C++	Cahon et al. (2004)	<a href="http://paradisao.gforge.inria.fr">http://paradisao.gforge.inria.fr</a>
Symphony	✓	✓	MILP	exact	B&C	✓	✓	✓	C	Ralphs and Glazebrook (2006)	<a href="https://projects.coin-or.org/Symphony">https://projects.coin-or.org/Symphony</a>
Xpress-IVE	✓	✓	MILP/MIQP and others	exact	B&C	✓	✓	✓	C, C++, Java, Fortran, VB6 and .NET	-	<a href="http://www.fico.com/xpress">http://www.fico.com/xpress</a>



a comprehensive list of open source solvers see Linderoth and Ralphs 2004). Also, we listed AIMMS and GAMS because they can invoke user-provided algorithms via the Open Solver Interface (OSI) (Saltzman 2002; Lougee-Heimer 2003). Batch experiments can be implemented in CPLEX/OPL Studio, XPress-IVE, AIMMS, and GAMS by using their script languages.

The frameworks related the most closely to our work are EasyLocal++ (Di Gaspero and Schaerf 2003), EasyAnalyzer (Di Gaspero et al. 2007), jMetal (Durillo et al. 2006, 2010) and oMetah (see lower section of Table 1).

EasyLocal++ (Di Gaspero and Schaerf 2003) is an objected-oriented software framework supporting the implementation and analysis of local search algorithms/metaheuristics. It contains generic implementations of several algorithms (amongst others tabu search and simulated annealing) that can be specialized for an application by providing classes for problem instances, solutions, solution representations used within algorithms, and moves within local search neighborhoods.

EasyLocal++ includes so-called *testers* that, among other things, can be used to perform computational experiments sequentially in batch mode and collect data on experiment results. Experiments are specified using the proprietary script language EXPSPEC by describing a set of experimental units with different algorithm variants to be executed for a given problem instance, each repeated for a specifiable number of times. Testers generate plots showing computational results in an aggregated graphical representation. EasyLocal++ was implemented in C++ using the design patterns *template method* and *strategy method*.

EasyAnalyzer (Di Gaspero et al. 2007) is a C++ software framework for the experimental analysis of local search algorithms that has been integrated with EasyLocal++. In addition to algorithms implemented with Easy Local++, also algorithms implemented in other programming languages can be integrated into EasyAnalyzer as “black box” standalone executables invoked via a command-line interface. It offers the following experimental analysis functionality: (1) Analyze the search space for a specific local search neighborhood of a problem instance. (2) Analyze the run-time behavior of an algorithm. (3) Compare several algorithms and determine the best algorithm variant by applying a *racing procedure* that iteratively discards inferior algorithms.

jMetal (Durillo et al. 2006, 2010) is a Java software framework for implementing and analyzing metaheuristics for multi-objective (continuous, integer, or mixed-integer) optimization problems that already contains generic implementations of various multi-objective metaheuristics. jMetal comprises an experiments package with graphical user interface (GUI) that offers the following functionality: (1) Run batch experiments for a specified set of algorithm variants on a set of problem instances. (2) Automatically generate a LaTeX table containing values of quality indicators for each pair of algorithm and problem instance. (3) Run statistical tests for analyzing the significance of differences between algorithms. (4) Generate boxplots to visualize quality indicators.

oMetah<sup>13</sup> is a C++ software framework supporting the implementation and experimental analysis of metaheuristics for continuous optimization problems. Designed

<sup>13</sup> <http://ometah.berlios.de/>.

with a client-server architecture, it can be used to run experiments in parallel on a cluster of computers. It contains generic implementations of several meta-heuristics. oMetah can be used to perform batch computational experiments just as EasyLocal++/EasyAnalyzer. In contrast to the aforementioned frameworks, oMetah contains persistence functionality, which however stores results in a decentralized way: It saves a complete description of each experimental unit and its results in a separate Extensible Markup Language (XML)<sup>14</sup> file. The supplementary module oMetahLab can generate reports and plots from the produced XML files.

The Optimization Services (OS) project (Fourer et al. 2008) is a *cloud computing* approach for optimization: It is composed of a distributed infrastructure for optimization via the internet and a corresponding XML standard (the Optimization Services Protocol) for exchanging data. A *registry* (repository/directory) is used to provide meta-data on *optimization service providers* to optimization clients. However, the OS architecture does not offer persistence functionality for problems and solutions, which it delegates to the optimization clients. Also, its focus is on the production use rather than the development phase of algorithms.

### 3.2 The link between experimental algorithmics and OR software frameworks

OR software frameworks with experimental analysis functionality like OREX-J serve as tools for the experimental analysis of algorithms. However, the mere usage of a software framework in a research project does not ensure a well-designed experimental analysis. It eases the design, execution, and analysis of experiments and might remove some common sources of flaws. Yet, in addition, the methodology has to be sound. Even though almost every OR paper containing a new algorithm that is tested against a set of problem instances falls into the category experimental algorithmics, hardly any methodological papers on experimental algorithmics have been published in *OR journals*. In contrast, several publications of researchers from the *computer science community* address methodology aspects in the experimental analysis of algorithms: Stützle et al. (2009) and Birattari (2009) describe experimental methodology for assessing the performance of metaheuristics. Other publications on the subject include Johnson (2002), Demetrescu and Italiano (2000), Moret and Shapiro (2011), Moret (2002), Bartz-Beielstein (2003) and Taillard (2005).

Johnson (2002) provides a guideline on how to analyze algorithms experimentally by recommending best practice principles and describing typical flaws in computational studies as well as ways of resolving them.

Experimental analysis software frameworks such as OREX-J can help implement the following four aspects (aspects 2 and 3 are related to principles described by Johnson 2002):

1. “Variance reduction”: Generic support for variance reduction techniques like common random numbers (CRN) (Law 2006, p. 578ff.) can be built into a framework.
2. “Ensure reproducibility”: A generic persistence layer could help save output data of experiments containing all relevant information that might be needed now or

<sup>14</sup> <http://www.w3.org/XML/>.

later. Such information could describe the problem instance that was solved, the version of the algorithm, date and time when the experimental unit was executed, the computer on which it was run, and random seeds that were used for generating problem instances or in a randomized algorithm. Saving data on seeds of random numbers used when generating problem instances or running an algorithm and other details of an experimental unit is an important aspect of ensuring reproducibility of experiments that can be eased by a framework. This can be crucial for debugging an algorithm, or running additional experiments for a revision of a paper submitted some time ago.

3. “Ensure comparability”: Assuming that a user integrates her own algorithm into a framework that already contains several rival algorithms, she can easily compare the algorithms on one computer to ensure that differences in solution quality and run-times are not due to the fact that the authors of rival algorithms used slower or faster computers. Also, having problem instance generators and loaders and various algorithms integrated into a framework makes it easier to run different algorithms on the same set of problem instances.
4. “Backup strategy”: The probability of a loss of data on test instances or experiments due to technical problems or human errors can be reduced by a framework that saves all data into a central database backed up regularly.

### 3.3 Software engineering

Two topics in software engineering are closely related to OREX-J: *Design patterns* and *software testing*. Design patterns can be defined as “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” (Gamma et al. 1994, p. 3).<sup>15</sup>

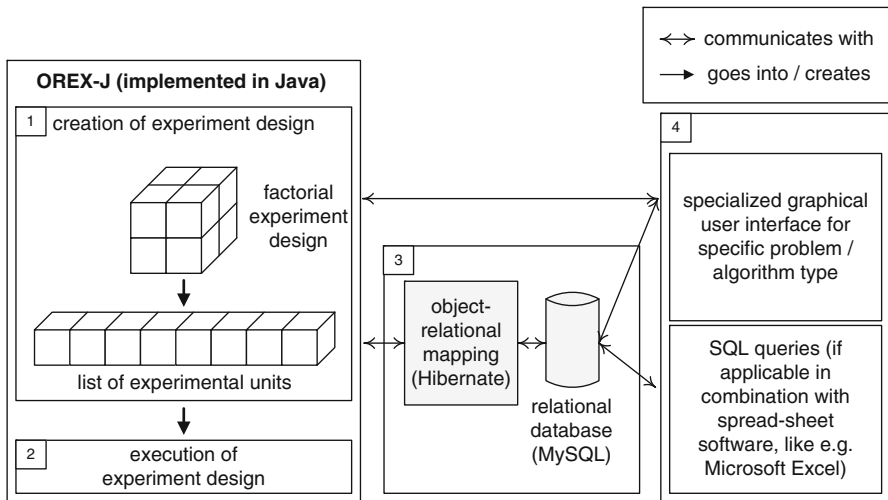
We used four design patterns in the implementation of OREX-J: Prototype, Factory Method, Template Method, and Singleton (Gamma et al. 1994).

“Prototype”: When employing the prototype pattern, objects are created by cloning a prototypical instance (instead of using a normal constructor and getter and setter methods to create a copy of an object). The *manipulators* (see Sect. 4.3.1) in OREX-J are an implementation of the prototype pattern.

“Factory Method”: This pattern uses a method (which is part of an abstract *creator* class) to create new objects. A concrete creator class has to be implemented by the user to determine the type of the objects that are created. We use this pattern in OREX-J to create problem instances and different variants of algorithms (see Sect. 4.3.2).

“Template Method”: Often, certain steps in several related algorithms are identical on an abstract level, i.e. there exists an invariant part for a family of algorithms. A template method defines the algorithm in terms of abstract operations that concrete classes implement. This template method that is part of an abstract class contains the invariant parts of the algorithm (which are therefore implemented only once) and

<sup>15</sup> Note that design patterns are independent of programming languages (PL) in the sense that they are abstract conceptual patterns that can be used in software design in various PL.



**Fig. 3** Architecture of a typical solution implemented with OREX-J

leaves the individual parts of the algorithm to the subclasses. This pattern is used in OREX-J to facilitate code reuse.

“Singleton”: Some classes should be only instantiated once during the execution of the program. The singleton pattern serves to ensure that there is at most one instance of a certain class. It specifies a defined point of access to a class by assigning the responsibility of managing the creation of and access to the single instance to the class itself. We e.g. employ the singleton pattern to enable that successive experiments use the same MILP solver library instance instead of unnecessarily creating a new one each time. Also, we use it to channelize access to the current Hibernate session.

Software testing methodology and in particular automated component testing (Spillner et al. 2007, p. 38ff, 97ff) is closely related with the implementation of optimization algorithms: Each experimental unit can be seen as a component test and often serves for precisely that purpose while an algorithm (the component) is being implemented and debugged. Automated test frameworks such as JUnit<sup>16</sup> exist, but in contrast to OREX-J, they can neither generate experimental designs ( $REQ_{A1}$ ) nor provide generic persistence ( $REQ_D$ ).

## 4 OREX-J architecture

### 4.1 Overview

A high-level overview of the architecture of OREX-J is given by Fig. 3. The usage of OREX-J can be categorized into four aspects: (1) Creation of experimental designs, (2) execution of experimental designs, (3) utilization of the persistence infrastructure, and (4) analysis of computational results. Aspects 1 and 2 are visualized in more

<sup>16</sup> <http://www.junit.org>.

detail by Figs. 5 and 6. OREX-J encompasses the concept of *full factorial designs* by providing functionality to simultaneously vary multiple *factors* such as algorithm and problem instance generator settings. Such factorial designs can be constructed using *manipulators* (see Sect. 4.3) and are automatically processed into a list of experimental units to be executed.

OREX-J was implemented in the OOP language Java.<sup>17</sup> In order to use OREX-J, the user has to implement eleven abstract Java classes/interfaces. Those abstract classes already contain *pre-defined* generic data structure and functionality.<sup>18</sup> However, the user needs Java programming skills to implement application-specific aspects (see Sect. 5 for examples).<sup>19</sup> The setup of computational experiments (with a number of experimental units) can be specified with few lines of code.

Algorithms implemented in other programming languages such as C++ can be integrated into OREX-J via the Java Native Interface (JNI).<sup>20</sup> The object-relational mapping (ORM) library Hibernate is used for providing persistence functionality, i.e. for storing information on experiments and their results in the RDBMS MySQL.<sup>21</sup> The logging library Log4J<sup>22</sup> is used for collecting and outputting status data while experiments are being executed.

It is possible to use spreadsheet software (e.g. Microsoft Excel) for analyzing computational results based on SQL queries that access the relational database. In addition, tailored GUIs that access computational results using OREX-J through Hibernate can be built for specific types of optimization problems. For instance, we developed a GUI for dynamic lot-sizing problems with substitutions (Lang and Domschke 2010) that visualizes data of problem instances and solutions as graphs and bar charts using the open source software libraries JUNG<sup>23</sup> and JFreeChart.<sup>24</sup>

As of now, one mixed-integer linear programming (MILP) solver (IBM ILOG CPLEX) has been integrated into OREX-J.

## 4.2 Data model

OREX-J is based on the generic data model depicted in Fig. 4: An *experimental design* generates an experiment consisting of a set of *experimental units*. Each experimental

<sup>17</sup> The reasons for choosing Java were as follows: Due to its ease of use and comfortable features such as garbage collection, Java is well-suited for developing algorithm prototypes. Also, Eclipse, a powerful integrated development environment (IDE) with comprehensive refactoring functionality, is available for Java as a free open source product. In terms of performance, Java has caught up on C++ due to hot spot compiler techniques, its performance is now comparable in many cases (Lewis and Neumann 2003).

<sup>18</sup> OREX-J consists of  $\approx 4000$  lines of code (LOC).

<sup>19</sup> For instance, the entire software implementation with OREX-J in Lang et al. (2008) consists of only  $\approx 2000$  LOC, of which less than 20% constitute framework-related “overhead”.

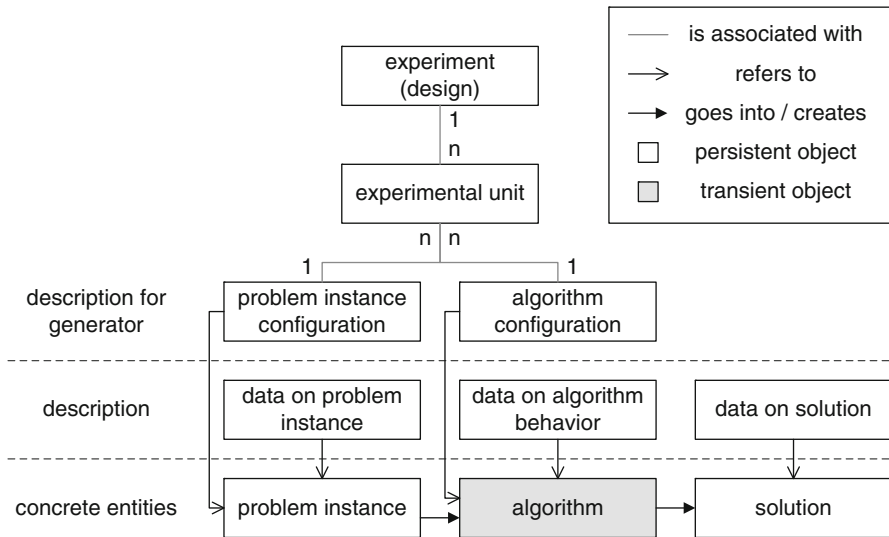
<sup>20</sup> [http://download.oracle.com/docs/cd/E17409\\_01/javase/6/docs/technotes/guides/jni/index.html](http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/technotes/guides/jni/index.html).

<sup>21</sup> We chose Hibernate as it is a well-documented, mature, easy-to-use ORM implementation. Note that the RDBMS does not need to be installed on the local machine, but could also reside elsewhere and be accessed via intra-/internet.

<sup>22</sup> <http://logging.apache.org/log4j/>.

<sup>23</sup> <http://jung.sourceforge.net>.

<sup>24</sup> <http://www.jfree.org/jfreechart/>.



**Fig. 4** Data model of OREX-J

unit refers to one *problem instance configuration* (describing an instance to be generated or a real-world instance) and one *algorithm configuration* to be used to solve the problem instance.

Referring to one problem instance, there is an object that contains data describing the problem instance such as *metrics* (e.g. the maximum distance between an existing warehouse and a customer in a supply chain design model).

An algorithm object receives a problem instance as its input and returns a solution to the problem instance. Also, it is associated to data describing the algorithm behavior, e.g. its run-time or the number of iterations of a metaheuristic such as tabu search. A solution is described by an object containing metrics (e.g. its objective value, or the capacity utilization resulting from a production plan) and other data (e.g. flags indicating whether the solution is feasible or infeasible, or violates certain soft constraints).

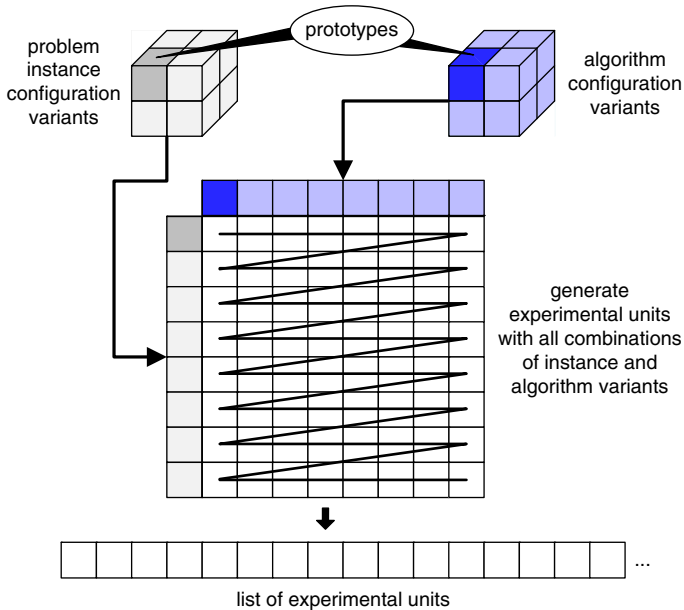
All entities represented by white boxes in Fig. 4 are persistent objects, i.e. they can be stored permanently in a relational database by means of the ORM. Thus, they can be accessed later for performing analyses of computational results.

The data model is illustrated in detail for the case study in Sect. 5.5.

### 4.3 Architecture in detail

#### 4.3.1 Creation of an experimental design

An experimental design is generated as depicted in Fig. 5: The OREX-J user provides prototype objects for the problem instance and algorithm configuration. Those objects serve as the “base case” experimental unit from which other experimental units are obtained by *manipulating* certain *factors*, i.e. by trying all possible combi-



**Fig. 5** Creation of an experimental design in OREX-J

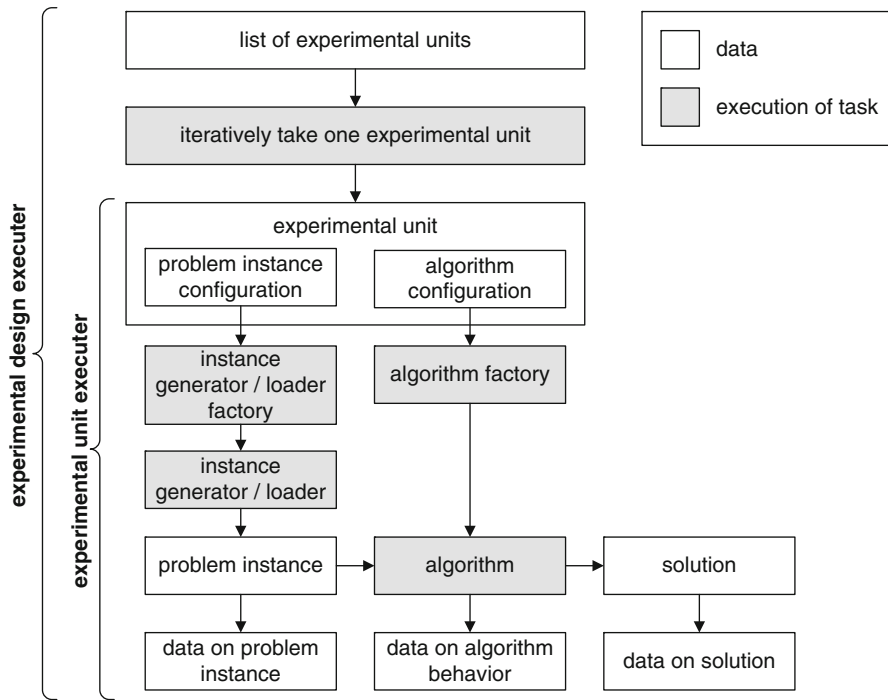
nations of different *levels* of certain attributes of the problem instance or algorithm configuration. These manipulations of attributes are performed by manipulator objects. A problem instance/algorithm manipulator receives a prototype object as its input and generates a list of problem instance/algorithm configurations, in each of which one or more attributes (factors) have been set to certain values (levels). The manipulators in OREX-J can be seen as an application of the prototype pattern (see Sect. 3.3).

Manipulators can e.g. be used to generate problem instance configuration variants with different sizes, or algorithm configuration variants with different run-time limits. When using e.g. three manipulators for generating problem instance variants, the resulting combinations can be visualized as a three-dimensional cube as shown in Fig. 5. Referring to a capacitated lot-sizing problem, the three factors could e.g. be the number of products, number of periods, and tightness of capacities, each with two possible values.

After generating “cubes” for problem instance and algorithm variants, those are transformed into variant lists. By iterating over all possible combinations of these variants, OREX-J generates a list of experimental units to be executed.<sup>25</sup>

<sup>25</sup> “Cube-like” full factorial designs may not be purposeful in all cases: For instance, if several metaheuristics have been implemented for a problem, they might have some algorithm configuration parameters (e.g. a run-time limit) in common, but each of them could have some special parameters that are not applicable to other metaheuristics (e.g. the population size of a genetic algorithm, or the tabu list length of a tabu search algorithm). Also, it might e.g. be the case that the user wants to use different run-time limits for different algorithms.

In such cases, OREX-J can be used in the way that a “cube” is generated for each class of algorithms (e.g. separately for each type of metaheuristic) or for each class of problem instances (e.g. separately for each problem instance generator if multiple are available). This results in “tree-like”, split experimental



**Fig. 6** Execution of an experimental design in OREX-J

#### 4.3.2 Execution of an experimental design

Figure 6 describes the process flow during the execution of an experiment (i.e. a set of experimental units) created based on an experimental design: An *experimental design executor* object takes a list of experimental units as its input and iteratively hands those to an *experimental unit executor* that execute the individual experimental units.<sup>26</sup>

The experimental unit executor proceeds as follows: It creates<sup>27</sup> a suitable *problem instance generator* or *loader*, which successively returns a problem instance as specified in the configuration. This problem instance contains a method that returns data describing itself. The algorithm configuration is passed over to an *algorithm factory*

Footnote 25 continued

designs as exemplified by Fig. 10, where each leaf represents a level of a factor and the nodes above those leafs represent factors (or sets of factors). Also, OREX-J includes a “compatibility check” that allows to exclude certain combinations of problem instances and algorithm configurations.

<sup>26</sup> Instead of executing the experimental units in the order of the list generated as portrayed in Fig. 5, we randomize their execution order by shuffling the list. The reasoning behind this is to reduce the effect of exogenous distortions of computational results, e.g. caused by virus scanners or operating system updates that temporarily decrease system performance while a subset of related experimental units is being executed.

<sup>27</sup> This happens by means of a problem instance generator/loader factory object that returns a problem instance generator/loader object.



that creates an algorithm object as specified. This algorithm returns a solution to the problem instance and generates data describing its behavior (see Sect. 4.2).<sup>28</sup>

OREX-J uses the template method pattern in several spots, e.g. by defining an algorithm interface with a method named `solve` to be overridden by users.

After the execution of each experimental unit, its results are stored via Hibernate and can be accessed from this point in time on (e.g. via remote database access).

OREX-J contains an *e-mail notification* feature that sends a message to a specified address after an experiment has been executed. This avoids inconvenient, repetitive manual checks to see whether experiments have completed that interrupt a researcher's flow of work.

#### 4.3.3 Persistence infrastructure for experimental designs and results

The persistence of experimental designs and results was implemented using the ORM library Hibernate and its extension Hibernate Annotations.<sup>29</sup> Using Hibernate Annotations, the ORM of Java classes to relational database tables can be specified on the spot in the Java source code by adding a comparatively small number of lines, rather than providing separate mapping files in addition to the Java files. Hibernate *automatically* generates all required tables in the RDBMS.<sup>30</sup>

OREX-J contains a pre-defined ORM for all persistent classes in Fig. 4 that includes the relationships between those classes with the respective cardinalities. All of those persistent entities except for the experimental design and experimental unit, which are generic classes, need to be implemented in an application-specific way. This is done by inheriting from the corresponding abstract class, adding the required fields and functionality, and specifying an ORM for its fields using Hibernate Annotations (for details see Sect. 5).

The structure of an experimental design is stored implicitly by saving the problem instance and algorithm configuration for each experimental unit.

#### 4.3.4 Analysis of computational results

Once an experiment has been executed, the relational database contains all persistent entities of the data model (see Fig. 4) characterizing the experiment and its results. Those data can be accessed in three ways: (1) Via SQL using spreadsheet software that provides pivot table functionality for aggregating results and visualization functionality (charts), (2) via SQL using a relational database GUI (e.g. Microsoft Access), or (3) via Hibernate in a specialized, application-specific GUI implemented in Java. Option 3 makes use of the possibility to retrieve previously saved objects (e.g. solutions or data on algorithm behavior) from the relational database via the ORM.

<sup>28</sup> The problem instance generator/loader, problem instance generator/loader factory, and algorithm factory are implementations of the factory method pattern (see Sect. 3.3).

<sup>29</sup> [http://docs.jboss.org/hibernate/stable/annotations/reference/en/html\\_single/](http://docs.jboss.org/hibernate/stable/annotations/reference/en/html_single/).

<sup>30</sup> Due to technical restrictions of Hibernate, it was not possible to directly map Java interfaces to tables in the relational database. Hence, we had to use abstract classes instead of interfaces for persistent application-specific entities that have to be implemented by the user by inheriting from those abstract classes.

A key advantage of the OREX-J approach of storing all data in one place in a relational database over using log files (“quick-and-dirty” approach) is that it directly enables “massive number crunching” by means of simple SQL queries. In contrast to log files, data are stored in a manner that is structured per se.

#### 4.4 Reproducing randomness in computational experiments

Randomness is used for two purposes in computational experiments: (1) For generating random problem instances and (2) for stochastic algorithms (i.e. algorithms with nondeterministic behavior). As argued in Sect. 3.2, ensuring the reproducibility of (pseudo-)randomness in computational experiments can be essential for effectively debugging an algorithm, or reconstructing results of a past experiment. Initializing a random number generator using the current system time as its seed clearly hinders reproducibility.

Hence, we decided to build a mechanism for reproducing randomness into OREX-J: The problem instance and algorithm configuration each contain a so-called *meta-seed* field. The problem instance configuration meta-seed is used to initialize a “meta random number generator” that creates seeds for various random number generators required for generating parameters of the problem instance (purpose 1), e.g. differently distributed demand time series of products in a production planning model. The algorithm meta-seed is used to initialize a random number generator that creates seeds for random number generators required for implementing randomized behavior in stochastic algorithms (purpose 2). By this mechanism, the entire pseudo-randomness of an experimental unit is captured by the two meta-seeds in the problem instance and algorithm configuration, eliminating any irreproducible randomness.

One consequence of this mechanism was that we could easily integrate CRN in OREX-J: For instance, one can reduce variance in computational results by using the same meta-seed for generating problem instance variants that should only differ in a small trait but have all other data in common.

#### 4.5 Applications

OREX-J has already been employed for both deterministic and stochastic models and heuristic as well as exact algorithms. In the following, we sketch three research areas where OREX-J has been used in practice:

1. [Lang and Domschke \(2010\)](#) compare different reformulations for MILP models for *dynamic lot-sizing with substitutions* on generated problem instances using OREX-J. [Lang and Shen \(2011\)](#) develop two MILP-based heuristics for a capacitated lot-sizing and scheduling model with sequence-dependent setups and substitutions and use OREX-J to compare those in computational experiments.
2. [Lang \(2008\)](#) considers a *simulation-based optimization approach* for a stochastic multi-location blood bank inventory control model based on a discrete-event simulation. A heuristic for improving inventory control policy parameters is examined for different policy variants using OREX-J.

3. [Lang et al. \(2008\)](#) develop a MILP model for *software supplier selection and product portfolio* planning that includes substitutions of software components. OREX-J is used for computational experiments that analyze the characteristics of optimal solutions for different scenarios, employing CPLEX for solving problem instances.

## 5 Case study: dynamic lot-sizing with substitutions

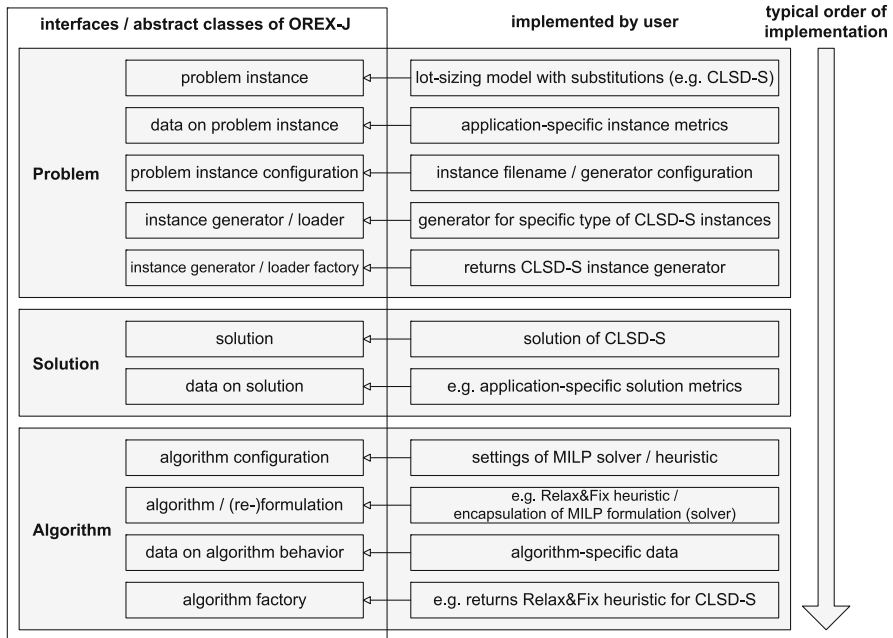
This section illustrates the usage of OREX-J in a case study based on the aforementioned studies [Lang and Domschke \(2010\)](#) and [Lang and Shen \(2011\)](#) that compare reformulations and algorithms for dynamic lot-sizing models with substitutions in computational experiments. It explains how to (1) define an optimization problem and implement algorithms and reformulations, (2) perform necessary application-specific implementations of interfaces and abstract classes, (3) create and execute an experimental design, and (4) analyze computational results. Also, the resulting OREX-J data model is described in detail in Sect. 5.5.

### 5.1 Optimization problems, algorithms, and reformulations

In this case study, we describe how to use OREX-J for computational experiments to compare reformulations and algorithms for deterministic, single-level, multi-product, multi-period dynamic lot-sizing models with substitutions. In this context, substitution means that a product can be replaced by certain others. The Lot-Sizing Problem with Substitution and Initial Inventory (LSP-SI) is uncapacitated, whereas the Multi-Resource Capacitated Lot-Sizing Problem with Substitution (MR-CLSP-S) extends the LSP-SI by introducing multiple capacitated production resources with time-varying capacities and fixed production setup times ([Lang and Domschke 2010](#)). The Capacitated Lot-sizing problem with Sequence-Dependent setups and substitutions (CLSD-S) ([Lang and Shen 2011](#)) is an extension of the LSP-SI that assumes a single capacitated production resource, sequence-dependent setup costs and times, backloging and overtime. An LSP-SI formulation is given in Appendix B.

The common assumptions of those models are as follows: Initial inventories of the products are in stock at the beginning of the first period. One quantity unit of a product satisfies demand of exactly one quantity unit of a demand class, i.e. the substitution ratio is 1:1. Only certain products can be used to satisfy demand of a specific demand class. Conversion of units of a product is carried out immediately before shipping to the customer, thus no converted goods are kept in stock. The decision variables model setup and lot size decisions and the allocation of stocks to demands, i.e. the substitution decisions. The objective is to minimize the sum of fixed setup, variable production, holding, and substitution costs.

[Lang and Domschke \(2010\)](#) develop Simple Plant Location (SPL)-based reformulations for the LSP-SI and MR-CLSP-S to obtain tighter lower bounds and reduce MILP solver run-times. [Lang and Shen \(2011\)](#) develop MILP-based Relax&Fix and Fix&Optimize heuristics for the CLSD-S and compare different variants of those in computational experiments.



**Fig. 7** Example—usage of OREX-J for analyzing algorithms for a capacitated lot-sizing problem

## 5.2 Implementing application-specific classes and defining persistence

It is necessary to implement 11 classes before defining an experimental design for a dynamic lot-sizing model with substitutions, which are summarized in Fig. 7:

1. **Problem instance:** It contains fields for all problem data (e.g. demand data, setup costs, holding costs, etc.).
2. **Data on problem instance:** It contains application-specific data describing the problem instance (e.g. a metric that measures the scarceness of production capacities or the average number of substitutes available for a product).
3. **Problem instance configuration:** It provides the problem instance generator/loader with data that unambiguously specifies which type of problem (e.g. whether the problem instance should have uncapacitated or capacitated production resources, and whether backlogging and overtime are allowed) and how it should be generated (e.g. the number of products and periods, the substitution structure, the levels of costs, demand pattern, etc.). This problem instance configuration can also serve as an input for a problem instance loader that loads an existing (real-world) instance from a database or file. In that case, the problem instance configuration would contain the name and location of the problem instance.
4. **Problem instance generator/loader:** It creates a problem instance based on a problem instance configuration or loads an existing instance specified in the instance configuration. When loading instances rather than generating them, there is no restriction on the data format of those instances. However, as there is no generic

data format for instances of all types of optimization problems, users have to implement a specific instance loading mechanism for each problem type. Once loaded, an instance can be saved in the relational database via the ORM.

5. Problem instance generator and loader factory: It returns a suitable problem instance generator or loader for the problem type specified in the problem instance configuration.
6. Solution: This class contains fields for the values of all decision variables (e.g. lot-sizes, binary variables that indicate setups, inventory levels, and substitution quantities) as well as methods that, e.g., determine whether a solution is feasible, calculate its objective value, or check whether the solution requires backlogging or overtime.
7. Data on solution: It contains application-specific data describing a solution (e.g. a flag that indicates whether the solution is feasible, its objective value, beta service level, and capacity utilization).
8. Algorithm configuration: It contains an unambiguous description of the algorithm variant or reformulation to be used in an experimental unit (e.g. whether a standard MILP B&C solver should be used with the original formulation or a reformulation, or a Relax&Fix heuristic) as well as algorithm or reformulation parameters (e.g. a time limit, Relax&Fix time window size, or a value parameterizing an approximate extended formulation).
9. Algorithm/(re)-formulation: Such classes implement specific algorithms or formulations, e.g. a Fix&Optimize heuristic, a standard MILP formulation, or a SPL-based formulation for the lot-sizing problem.
10. Data on algorithm behavior: It contains application-specific data describing the algorithm's behavior (e.g. its run-time, the relative MILP gap upon termination, the number of iterations performed, or the reason for termination (optimal solution found, problem infeasible, or time limit exceeded)). These data may include information on the algorithm progress over time, such as the objective value of the best known solution and optimality gap at different points in time, as well as corresponding solutions.
11. Algorithm factory: It creates an algorithm object conforming to a given algorithm configuration object.

Those classes can be implemented by inheriting from the corresponding abstract framework classes (see Sect. 4.3.3). The persistence of the problem instance configuration is defined using Hibernate Annotations as shown exemplarily in Listing 1. No “intervention” is required for simple parameters such as the number of periods, whereas the correct mapping has to be defined for enumeration type parameters as well as arrays.<sup>31</sup>

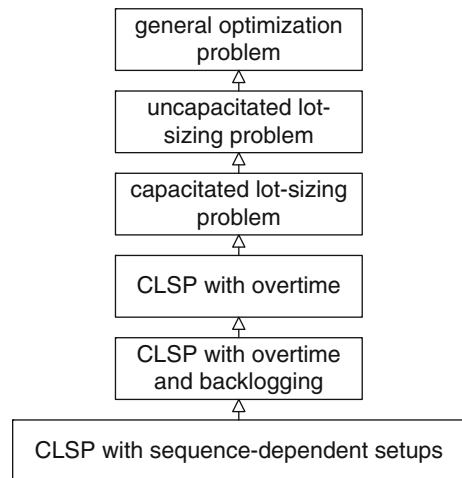
<sup>31</sup> Due to technical restrictions of Hibernate, arrays such as production quantity matrices of a lot-sizing model are not automatically mapped as persistent data. However, such arrays contained in problem instances or solutions can be stored by converting them into Java map objects (associative arrays) ([http://download.oracle.com/docs/cd/E17409\\_01/javase/6/docs/api/java/util/Map.html](http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/api/java/util/Map.html)) A map contains one key-value pair for each array field, where the key object contains the indices of the field (e.g. the product and period index) and the value object the corresponding value contained in the field (e.g. a production quantity). It is not required to use this representation inside of the algorithm, as it can use a specialized, internal solution representation.

**Listing 1** Defining persistence of problem instance configuration

```

1  @Entity
2  @Table(name = "LSPSIGeneratorConfig")
3  public class LSPSIGeneratorConfig extends
      AbstractInstanceGeneratorConfig {
4      ...
5      private int nPeriods;
6      ...
7      @Enumerated(EnumType.STRING)
8      @Column
9      private InitialInventory initialInventory =
          InitialInventory.NO;
10     ...
11 }

```

**Fig. 8** Usage of inheritance for generalizations of lot-sizing model with substitutions

An existing OREX-J database schema for a lot-sizing model can be easily extended by additional attributes such as new solution metrics or algorithm configuration data by adding those to the corresponding classes.<sup>32</sup>

One main advantage of implementing the entities of the OREX-J data model (see Sect. 4.2) in the OOP language Java was that we could use the inheritance principle of OOP. Rather than using a “copy-paste approach” for developing generalizations of a basic lot-sizing model with substitutions (e.g. 10 model variants implemented in a MILP modeling language kept and maintained in different files), we let generalized classes inherit from the classes for simpler model variants. This inheritance approach aims at increasing source code *reuse* and improving *maintainability* and *extensibility*. It is illustrated by Fig. 8, which refers to problem instance

<sup>32</sup> For instance, a new solution metric can be added by inserting it as a new attribute in the “data on solution” class. New algorithm types or parameters can be added by inserting new attributes in the “algorithm configuration” class.

classes: The problem instance of our basic LSP-SI inherits from a generic, abstract optimization problem class of OREX-J. Its generalization with capacitated production resources (MR-CLSP-S) inherits from that basic class, and only adds additional required data, i.e. the available capacities and production setup times. An extension of the MR-CLSP-S by allowing for overtime production only requires one additional field containing the overtime costs to be incorporated in the objective function. In order to extend the variant with overtime by backlogging, one can inherit from that class and add a field containing the backlogging cost data. If we want to generalize that variant to include sequence-dependent setup costs and times (which results in the multi-resource CLSD-S variant), we can inherit from it and add fields containing data on sequence-dependent setup costs and times. When implementing algorithms for these model variants, we repeatedly used the template method pattern, e.g. to define the setup cost term differently for models with sequence-dependent setups by overriding a method that returns the total setup cost of a solution.

We implemented the classes in this inheritance structure in a way that the additional features in generalizations (e.g. overtime, backlogging, or setup times) can be switched on and off, e.g. to obtain a CLSD-S variant with backlogging and setup sequence-dependent costs, but no overtime and zero setup times.<sup>33</sup>

Similarly, one can create inheritance structures for other application-specific OREX-J classes, namely data on problem instances, algorithms, solutions, data on algorithm behavior, and data on solutions.

Implementing application-specific classes (e.g. for a lot-sizing model) with OREX-J facilitates code reuse among researchers: Different research groups could share the same classes, amongst others for instance generators, problem instances, solutions and algorithms in order to easily compare their algorithms with competing algorithms, or test their algorithms on problem instances generated by problem instance generators implemented by other researchers.

### 5.3 Creating and executing an experimental design

Listing C.1 contains a Java source code snippet that illustrates how to define an experimental design for reformulations and algorithms for the considered lot-sizing problem.

### 5.4 Analyzing computational results

Using OREX-J, the results of experiments can easily be aggregated and analyzed by comparatively simple and compact SQL queries run from within a spreadsheet software or database GUI. Also, it is possible to implement tailored GUIs in Java that directly access experiments and results via the ORM.

<sup>33</sup> A design alternative to using inheritance would be to use the decorator pattern (Gamma et al. 1994) for adding various generalizations to a lot-sizing model problem instance object: Each additional assumption such as backlogging or overtime would represent a decorator class, and those decorators could be stacked on top of each other.

We earlier mentioned the difficulty occurring in the “quick-and-dirty” approach that time-consuming experiments have to be repeated if a piece of information that would be very helpful for the analysis of the computational experiment has not been collected/calculated and stored in output files. With OREX-J, this difficulty can be addressed in two ways: (1) One can derive the required additional data from the database by means of a SQL query that e.g. calculates a new metric based on the solution representation in the database. However, this still cannot access *novel* properties from the algorithms (e.g. the iteration when the best solution was found—if this property has not been stored previously). In such cases only the second variant can be used to derive the needed information: (2) One can re-load a problem instance and the corresponding solution into Java using the ORM, determine the required additional data and write it into the RDBMS by adding an attribute to the corresponding class (e.g. a new metric added to the data on a solution).

#### 5.4.1 Using SQL queries and spreadsheet software

Listing C.2 shows a generic SQL statement that creates a view which contains the objective value of the best known solution for each problem instance contained in the database.

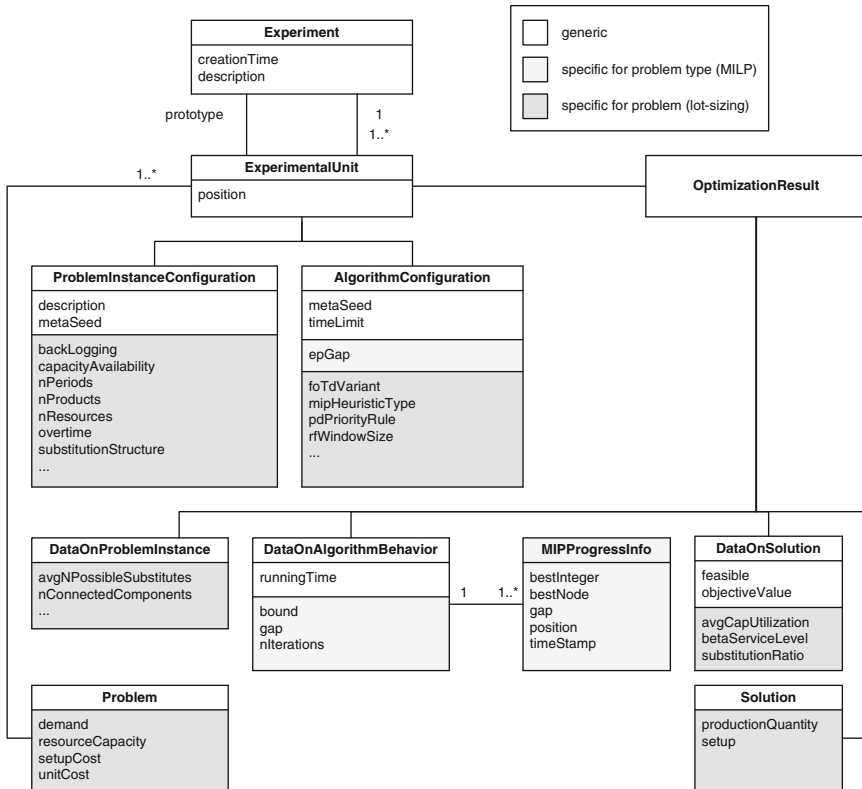
This is achieved by using the MIN aggregate function in combination with a GROUP BY statement. Performing a similar analysis based on log files and spreadsheets might require significant manual work for each new experiment, whereas this view is updated automatically once created. The definition of such views is enabled by the centralized storage of OREX-J that other frameworks (e.g. oMetah) lack. Listing C.3 presents an SQL query that performs a complex analysis comparing different algorithm variants for the lot-sizing model. For each pair of problem instance variant and algorithm variant, it determines four metrics by averaging over the 20 replications using the keyword GROUP BY and the aggregate function AVG: (1) the average capacity utilization (2) beta service level, (3) relative deviation from the objective of the best known solution, and (4) relative MILP gap calculated from the sharpest known lower bound.

Visualizations and aggregations of such analyses (e.g. bar diagrams) can easily be created by means of the pivot table and charting functionality of spreadsheet software. Similarly, progress diagrams showing the development of the best known solution (and/or lower bound) for an algorithm over time can be generated using SQL queries that access algorithm progress information stored in the database.

#### 5.4.2 Developing specialized user interfaces

We developed a tailored GUI for lot-sizing models with substitutions (Lang and Domschke 2010; Lang and Shen 2011) to examine the generated problem instances and obtained solutions graphically. Several screenshots of this GUI are shown in Appendix D.





**Fig. 9** (Modified) UML class diagram of the resulting data model for the lot-sizing case-study

## 5.5 Detailed illustration of data model

Figure 9 shows a detailed view of the data model for the lot-sizing case study from a database modeling point of view. The diagram is a modified Unified Modeling Language (UML)<sup>34</sup> class diagram structured analogously to Fig. 4.

Each box corresponds to one table in the relational database. Each of these tables corresponds to a Java class of OREX-J and all its subclasses. For instance, the box **AlgorithmConfiguration** corresponds to the generic algorithm configuration class, its subclass for MILP problems, and a subclass of the latter that contains algorithm parameters specific to lot-sizing algorithms.

Generic algorithm configuration attributes are shown in white boxes. Subclass attributes are shown in shaded boxes, where attributes specific for a certain problem type (e.g. MILP) are shown light gray and those specific for a problem (e.g. a lot-sizing problem) in dark gray. The letters “...” mean that additional attributes were omitted for reasons of space.

<sup>34</sup> <http://www.uml.org/>.

Edges between boxes represent relationships between classes/tables. The numbers annotated next to the edges specify the cardinality of those relationships. For instance, one or more experimental units are associated with an experiment and exactly one experiment is associated with each experimental unit. In addition, one prototype experimental unit is associated with each experiment (see Sect. 4.3.1). Edges without cardinality numbers stand for a relationship with 1:1 cardinality.

## 6 Conclusions and future research

We see four purposes of the framework OREX-J: (1) It can be used to reduce manual work of conducting computational experiments in various other research projects. (2) It allows users to apply en passant established paradigms known from literature on experimental algorithmics and on software patterns. (3) It simplifies the comparison of algorithms developed by different research groups: Instead of integrating other researchers' algorithms into proprietary, self-developed test beds, OR scientists could reuse OREX-J as a common framework for their experiments. (4) It can serve as a reference architecture and inspiration for other OR scientists who intend to implement their own experiment test beds (alternative 2 in Fig. 2).<sup>35</sup>

OREX-J fulfils the requirements  $REQ_A$ – $REQ_G$  stated in the introduction as follows:  $REQ_A$  is fulfilled by the mechanism shown in Fig. 6. OREX-J is not restricted to a specific type of problems or algorithms because of its generic data model illustrated in Fig. 4 ( $REQ_B$  and  $REQ_C$ ). The object-relational mapping Hibernate in combination with a relational database provides a centralized data storage ( $REQ_D$ ). OREX-J is available as free open source software ( $REQ_E$  and  $REQ_F$ ). It uses Java, a modern, popular object-oriented programming language ( $REQ_G$ ).

The core idea of OREX-J is to use a generic data model for storing experimental unit descriptions and results persistently in a relational database by means of an object-relational mapping. This enables the usage of SQL queries for executing complex analyses on computational results. Several object-oriented design patterns were employed in its design. OREX-J is available as open source software under the Apache License 2.0 via SourceForge. We hope that this will facilitate the continuous development and extension of OREX-J as an open source project by a community of researchers. There are basically five potential points of criticism:

1. “The framework is too complicated and over-engineered”: This may hold true for experiments that are not very complicated, rarely repeated or demand only a low degree of flexibility (in terms of the problem instances, algorithms, and reformulations to be analyzed). Based on our (limited) experience, the time savings for the evaluation of experiments, the increased flexibility and the convenience may outweigh the one-time effort of becoming acquainted with OREX-J in most cases. Nevertheless, each researcher has to evaluate appropriateness (i.e., the cost-benefit trade-off) for the used methods and tools in each research project.

<sup>35</sup> In addition, several traits of its design, in particular its persistence infrastructure, could be useful for the design of real-world OR software systems as well.

2. “The framework is not applicable for problem X or algorithm Y”: Due to its generic data model, OREX-J can be used for virtually any type of problem and algorithm. Even online and simulation-based optimization algorithms can be integrated.
3. “Why not use an existing software framework?”: None of the frameworks evaluated in the literature covers all of the requirements  $REQ_A$ – $REQ_G$  described in the introduction.
4. “Why not use a MILP solver GUI and scripting language (e.g. CPLEX/OPL Studio, Xpress-IVE, GAMS, or AIMMS) instead of OREX-J?”: Such standard software lacks a generic data model and persistence functionality. Also, one can e.g. combine OREX-J with Xpress by defining an optimization model in a .mos file and invoking the Xpress solver via its API.
5. “C++ is the programming language of first choice for implementing OR methods”: Algorithms implemented in other programming languages can be integrated via the JNI.

We see several next steps and research opportunities:

- Integrate MILP solvers other than CPLEX in the framework
- Integrate OREX-J with a *universal open source MILP solver interface* for Java (which does not exist yet but could be realized analogously to the Open Solver Interface (OSI)<sup>36</sup> for C++ or the commercial OptimJ<sup>37</sup>)
- Extend OREX-J by integrating metaheuristics frameworks (see Sect. 3.1).
- Adapt the OREX-J approach for *production use in industrial optimization software/decision support systems*, where it is often necessary to run, analyze and document many different *scenarios* and managerial decision restriction sets.
- Develop a *high-level graphical programming language* and user interface for specifying, running and analyzing experiments (similarly to the evolution from hand-coded assembler/C algorithms to high-level optimization modeling languages like AMPL/GAMS/AIMMS). This approach could be combined with *interactive visualization software for analytic data sets* (see the Tableau<sup>38</sup> system and Stolte et al. 2008).
- Combine the ideas contained in OREX-J with a *cloud computing* approach with distributed storage of problem instances and experiment results and distributed execution of experiments, possibly using NoSQL technology (Stonebraker 2010). For instance, one could integrate OREX-J into the Optimization Services architecture (Fourer et al. 2008).

**Acknowledgments** We would like to thank the editor and the two anonymous referees for their numerous helpful suggestions that substantially improved the manuscript. Thomas Widjaja’s research was supported by a research grant from the FAZIT-Stiftung.

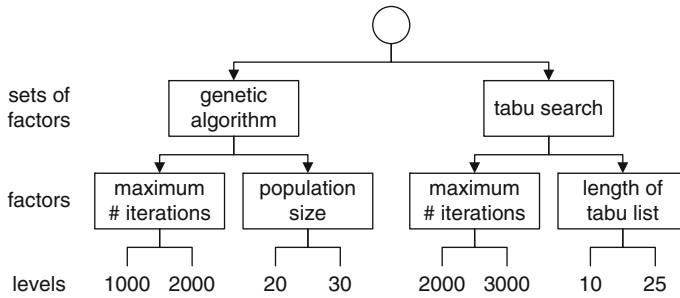
## Appendix A: Example: “Tree-like” experimental design

See Fig. 10.

<sup>36</sup> <https://projects.coin-or.org/OSi/>.

<sup>37</sup> <http://www.ateji.com/optimj.html>.

<sup>38</sup> <http://www.tableausoftware.com/public/>.



**Fig. 10** Example of a “tree-like” experimental design regarding algorithm variants

## Appendix B: Mathematical formulation of LSP-SI

Using the notation given in Table 2, the LSP-SI can be formulated as a linear mixed-integer programming model as follows:

$$\text{Minimize } F(q, x, I, s) = \sum_{i \in P} \sum_{t=1}^T \left( f_{it}x_{it} + p_{it}q_{it} + h_{it}I_{it} + \sum_{j \in D_i} r_{ij}s_{ijt} \right) \quad (\text{B.1})$$

**Table 2** Notation for the LSP-SI model

type	Symbol	Definition
Number of ...	$m$	Number of products
	$n$	Number of demand classes
	$T$	Number of periods
Indices and sets	$i \in P = \{1, \dots, m\}$	Products
	$j \in D = \{1, \dots, n\}$	Demand classes
	$t = 1, \dots, T$	Periods
	$V = P \cup D$	Vertex set of substitution graph
	$E \subseteq P \times D$	Directed edges of substitution graph denoting feasible substitutions: $(i, j) \in E$ if product $i$ can fulfil demand of class $j$
	$G = (V, E)$	Substitution graph
	$D_i = \{j \mid (i, j) \in E\}$	Set of demand classes whose demand can be fulfilled by product $i$
Parameters	$P_j = \{i \mid (i, j) \in E\}$	Set of products that can fulfil demand of class $j$
	$d_{jt}$	Demand of class $j$ in period $t$
	$h_{it}$	Non-negative holding cost for storing one unit of product $i$ in period $t$
	$p_{it}$	Unit production cost of product $i$ in period $t$
	$r_{ij}$	Substitution cost for fulfilling demand class $j$ by product $i$ per unit

**Table 2** continued

type	Symbol	Definition
Variables	$I_{i0}$	Initial inventory of product $i$
	$f_{it}$	Fixed setup or order cost for product $i$ in period $t$
	$q_{it}$	Production or order quantity of product $i$ in period $t$
	$s_{ijt}$	Quantity of product $i$ used to fulfil demand of class $j$ in period $t$
	$I_{it}$	Inventory of product $i$ at the end of period $t$
	$x_{it}$	Binary variable that indicates whether a setup for product $i$ occurs in period $t$

subject to

$$I_{it} = I_{i,t-1} + q_{it} - \sum_{j \in D_i} s_{ijt} \quad i \in P, t = 1, \dots, T \quad (\text{B.2})$$

$$d_{jt} = \sum_{i \in P_j} s_{ijt} \quad j \in D, t = 1, \dots, T \quad (\text{B.3})$$

$$q_{it} \leq M \cdot x_{it} \quad i \in P, t = 1, \dots, T \quad (\text{B.4})$$

$$q_{it} \geq 0, I_{it} \geq 0, x_{it} \in \{0, 1\} \quad i \in P, t = 1, \dots, T \quad (\text{B.5})$$

$$s_{ijt} \geq 0 \quad (i, j) \in E, t = 1, \dots, T \quad (\text{B.6})$$

The objective (B.1) is to minimize the sum of setup, unit production, holding and substitution costs. Equation (B.2) represents the *inventory balances*. Constraint (B.3) enforces that demand is always satisfied completely using the available substitution options (*product usage and substitution*). Due to constraint (B.4), quantity units of a product can only be purchased or produced in a period if setup takes place for the product in that period (*setup forcing*). (B.5)–(B.6) define the domains of the variables. The LSP-SI is NP-hard (Lang and Domschke 2010).

## Appendix C: Source code examples for case study

### C.1 Creating and executing an experimental design in Java

The experiment implemented in listing C.1 compares the original formulation with an SPL-based reformulation as well as a Relax&Fix heuristic (specified in l. 1–5) and tests two relative MIP gap cutoff values ( $10^{-4}$  and  $10^{-3}$ , specified in l. 7–11) for each of these three variants.<sup>39</sup> This results in a  $3 \times 2$  matrix of algorithm variants corresponding to the left cube in Fig. 5.

<sup>39</sup> In Relax&Fix, the relative MIP gap cutoff value specifies the termination criterion for each subproblem.

**Listing C.1** Lot-sizing problem experimental design implemented in Java using OREX-J

```

1  algGen = new VariousAlgorithmsGenerator();
2  algGen.addAlgorithm(cplex);
3  algGen.addAlgorithm(cplex_SPL_Reformulation);
4  algGen.addAlgorithm(relaxAndFix);
5  algorithmListGenerators.add(algGen);

6
7  List<Double> epGaps = new ArrayList<Double>();
8  epGaps.add(10e-4);
9  epGaps.add(10e-3);
10 VariousRelativeMIPGapsGenerator vEpGapG = new
    VariousRelativeMIPGapsGenerator(epGaps);
11 algorithmListGenerators.add(vEpGapG);

12
13 standardInstance.setCapacitated(true);
14 standardInstance.setProductionSetupTimes(true);
15 standardInstance.setNResources(1);

16
17 instanceListGeneratorsNoCRN.add(new
    VariousNProductsDemandClassesGenerator(8,16,4));
18 instanceListGeneratorsNoCRN.add(new
    VariousNPeriodsGenerator(8,12,2));

19
20 List<InitialInventory> ii = new ArrayList<
    InitialInventory>();
21 ii.add(InitialInventory.ZERO);
22 ii.add(InitialInventory.PERCENTAGE_OF_DEMAND_X_P);
23 standardInstance.setInitialInventoryPercOfDemand(0.10);
24 instanceListGeneratorsCRN.add(new
    VariousInitialInventoryGenerator(ii));

25
26 List<Double> ca = new ArrayList<Double>();
27 ca.add(1.05);
28 ca.add(1.1);
29 instanceListGeneratorsCRN.add(new
    VariousCapAvailabilityGenerator(ca));

30
31 instanceListGeneratorsNoCRN.add(new RepeatGenerator(20))
    ;

32
33 executeExperimentalDesign(algorithmListGenerators,
    instanceListGeneratorsNoCRN,
    instanceListGeneratorsCRN, standardAlgorithm,
    standardInstance);

```

The considered test instances are single-resource capacitated problem instances with non-zero setup times (l. 13–15). We consider problem instances with 8, 12, and 16 products and 8, 10, and 12 periods (specified in l. 17–18, where the last parameter in each line is a “step size”). Also, we generate problem instances with zero initial inventories as well as problem instances that have approx.  $x = 10\%$  of a product’s total demand in the planning horizon as its initial inventory (l. 20–24). Also, we compare problem instances that differ in the tightness of production capacities by generating variants with an excess capacity of approx. 5 % and 10 % (l. 26–29). This results in a 4-dimensional hyper-cube of problem instance variants, for each of which 20 replications are created, each with a different meta-seed (l. 31).

**Listing C.2** SQL view that returns the best found objective value for each lot-sizing problem instance of an experimental design

```

1  CREATE VIEW bestobjective AS
2  SELECT o.collection_id, i.instance_id, i.metaseed, MIN(
      objectiveValue) AS bestObj
3  FROM optimizationexperimentalunit o, algorithmconfig s,
      optimizationresult r, problemstatistics p,
      solutionstatistics sos, algorithmstatistics a,
      instancegeneratorconfig i
4  WHERE o.algorithmCfg_id=s.algorithm_id AND r.
      experimentalunit=o.id AND sos.id=r.
      solutionStatistics AND r.algorithmStatistics=a.id
      AND i.instance_id=o.instanceGenCfg_id AND p.id=r.
      problemStatistics AND sos.feasible='Y'
5  GROUP BY i.instance_id
6  ORDER BY i.instance_id

```

**Listing C.3** SQL query that returns the average capacity utilization, beta service level, best objective deviation and relative MIP gap for each algorithm configuration ran on a set of lot-sizing problem instances

```

1  SELECT o.collection_id, i.nProducts, i.nPeriods, i.
      initialInventory, i.capacityAvailability, s.
      algorithm_id, AVG(sos.avgCapUtilization), AVG(sos.
      betaServiceLevel), AVG((objectiveValue-bo.bestObj)/
      bo.bestObj) AS 'AvgbestObjDev', AVG((objectiveValue-
      bb.bestBnd)/bb.bestBnd) AS 'AvgbestBoundDev',
2  FROM bestobjective bo, bestbound bb, algorithmstatistics
      a, instancegeneratorconfig i,
      optimizationexperimentalunit o, optimalsol os,
      problemstatistics p, optimizationresult r,
      algorithmconfig s, solutionstatistics sos,
      feasiblesol f
3  WHERE sos.feasible LIKE 'Y' AND o.algorithmCfg_id = s.
      algorithm_id AND r.experimentalunit = o.id AND sos.
      id = r.solutionStatistics AND r.algorithmStatistics
      = a.id AND i.instance_id = o.instanceGenCfg_id AND p
      .id = r.problemStatistics AND a.id = os.id AND f.id=
      r.solutionStatistics
4  AND i.instance_id=bb.instance_id AND i.instance_id=bo.
      instance_id
5  GROUP BY o.collection_id, i.nProducts, i.nPeriods, i.
      initialInventory, i.capacityAvailability, s.
      algorithm_id

```

Only a single line of source code is required to execute this experimental design (l. 33)

The usage of CRN is controlled by adding some of the problem instance manipulators to a “no CRN list” and others to a “CRN list”, with the following consequence: A different meta-seed is used for generating each of the 20 replications within a group

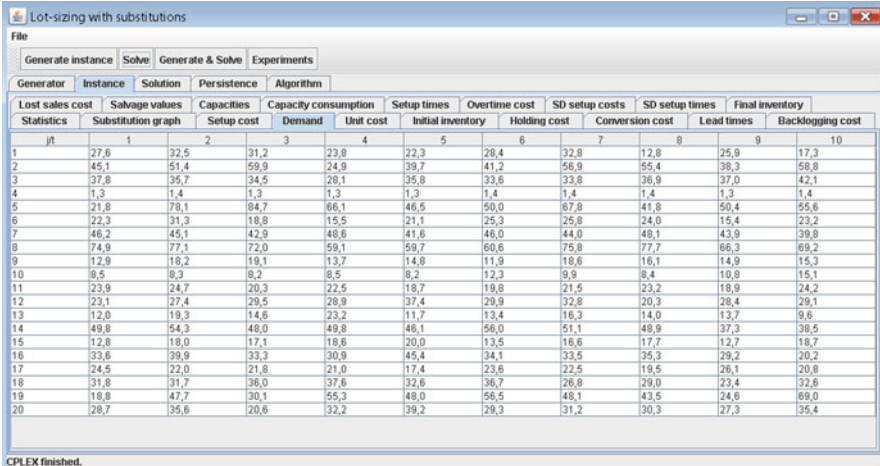
of problem instances a specific number of products and periods. However, the same meta-seed is used for generating problem instances that have the same number of products and periods, belong to the same replication, but have different initial inventory and capacity tightness parameters. Thus, those problem instances do not differ in the demand pattern and other problem data.

## C.2 Using SQL queries for analyzing computational results

Note that `o.collection_id` is the primary key identifying an experimental design.

## Appendix D: Screenshots of OREX-J-based GUI for lot-sizing with substitutions

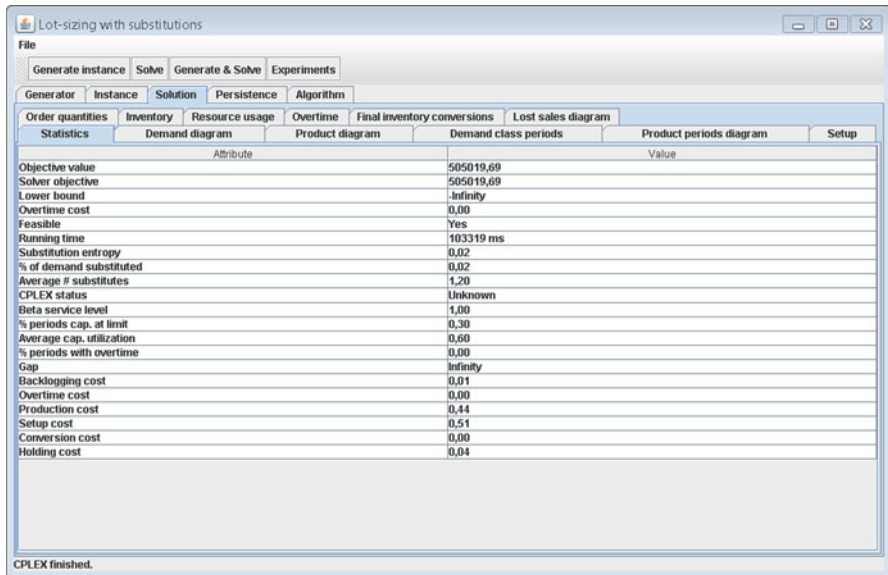
We implemented a tailored GUI to visualize and analyze problem instances, algorithm behavior and solutions for the lot-sizing models with substitutions. Figure 11 shows a screenshot of a table view of the demand data of a lot-sizing problem instance. Figure 12 displays an automatically generated table containing several metrics and other data describing the solution and the algorithm's behavior. Our GUI visualizes the substitution graph of a problem instance using the graph drawing library JUNG as shown in Fig. 13. The substitutions performed in a solution are visualized as a bar diagram (with one bar per demand class) using the charting library JFreeChart as depicted in Fig. 14.



	1	2	3	4	5	6	7	8	9	10
1	27.6	32.5	31.2	23.8	22.3	28.4	32.8	12.8	25.9	17.3
2	45.1	51.4	59.9	24.9	39.7	41.2	56.9	55.4	38.3	58.8
3	37.8	35.7	34.5	28.1	35.8	33.6	33.8	36.9	37.0	42.1
4	1.3	1.4	1.3	1.3	1.3	1.4	1.4	1.4	1.3	1.4
5	21.8	78.1	84.7	86.1	46.5	50.0	87.8	41.8	50.4	55.6
6	22.3	31.3	18.8	15.5	21.1	25.3	25.8	24.0	15.4	23.2
7	46.2	45.1	42.9	48.6	41.6	48.0	44.0	48.1	43.9	39.8
8	74.9	77.1	72.0	59.1	59.7	80.6	75.8	77.7	86.3	69.2
9	12.9	18.2	19.1	13.7	14.8	11.9	18.6	16.1	14.9	15.3
10	8.5	8.3	8.2	8.5	8.2	12.3	9.9	8.4	10.8	15.1
11	23.9	24.7	20.3	22.5	18.7	19.8	21.5	23.2	18.9	24.2
12	23.1	27.4	29.5	28.9	37.4	29.9	32.8	20.3	28.4	29.1
13	12.0	19.3	14.6	23.2	11.7	13.4	16.3	14.0	13.7	9.6
14	49.8	54.3	48.0	49.8	46.1	56.0	51.1	48.9	37.3	38.5
15	12.8	18.0	17.1	18.6	20.0	13.5	16.6	17.7	12.7	18.7
16	33.6	39.9	33.3	30.9	45.4	34.1	33.5	35.3	29.2	20.2
17	24.5	22.0	21.8	21.0	17.4	23.6	22.5	19.5	26.1	20.8
18	31.8	31.7	36.0	37.6	32.6	36.7	26.8	29.0	23.4	32.6
19	18.8	47.7	30.1	55.3	48.0	56.5	48.1	43.5	24.6	69.0
20	28.7	35.8	28.6	32.2	39.2	29.3	31.2	30.3	27.3	35.4

**Fig. 11** OREX-J-based GUI for lot-sizing with substitutions—table with demand data of instance

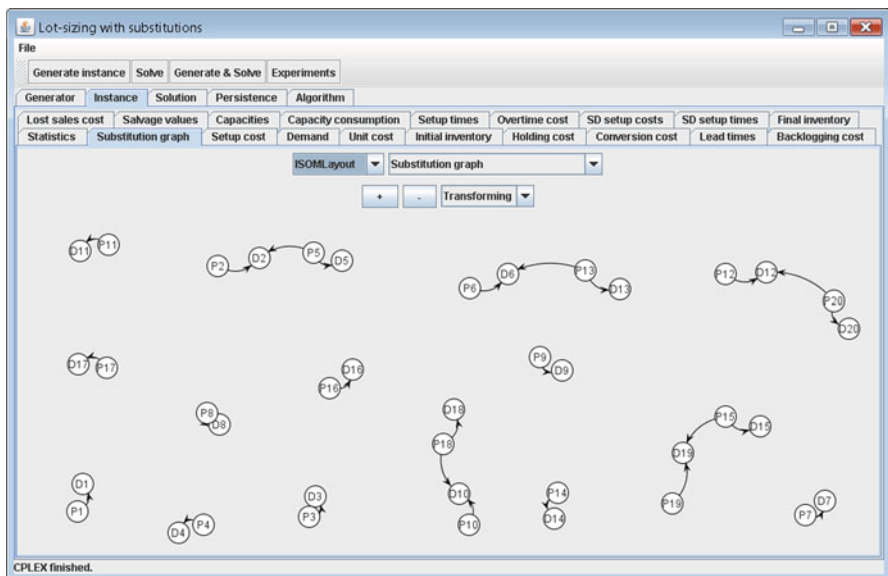




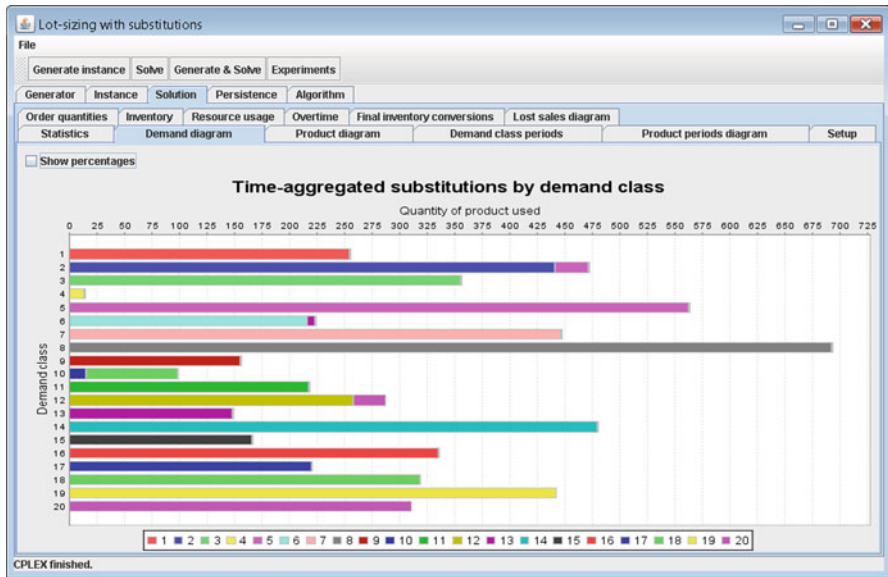
Attribute	Value
Objective value	505019,69
Solver objective	505019,69
Lower bound	Infinity
Overtime cost	0,00
Feasible	Yes
Running time	103319 ms
Substitution entropy	0,02
% of demand substituted	0,02
Average # substitutes	1,20
CPLEX status	Unknown
Beta service level	1,00
% periods cap. at limit	0,30
Average cap. utilization	0,60
% periods with overtime	0,00
Gap	Infinity
Backlogging cost	0,01
Overtime cost	0,00
Production cost	0,44
Setup cost	0,51
Conversion cost	0,00
Holding cost	0,04

CPLEX finished.

**Fig. 12** OREX-J-based GUI for lot-sizing with substitutions—statistics describing solution and algorithm behavior



**Fig. 13** OREX-J-based GUI for lot-sizing with substitutions—visualization of substitution graph of an instance



**Fig. 14** OREX-J-based GUI for lot-sizing with substitutions—visualization of substitutions performed in a solution

## References

- Bartz-Beielstein T (2003) Experimental analysis of evolution strategies—overview and comprehensive introduction. Technical report, Universität Dortmund, Dortmund
- Birattari M (2009) Tuning metaheuristics—a machine learning perspective. In: *Studies in computational intelligence*, vol 197. Springer, Berlin-Heidelberg-New York
- Cahon S, Melab N, Talbi E (2004) ParadisEO: a framework for the reusable design of parallel and distributed metaheuristics. *J Heuristics* 10(3):357–380
- Demetrescu C, Italiano G (2000) What do we learn from experimental algorithmics? In: Nielsen M, Rovay B (eds) *Mathematical foundations of computer science 2000 (MFCS 2000)*, Bratislava, Slovakia. *Lecture notes in computer science*, vol 1893. Springer, Berlin-Heidelberg-New York, pp 36–51
- Di Gasparo L, Schaefer A (2003) EasyLocal++: an object-oriented framework for the flexible design of local-search algorithms. *Softw Pract Exp* 33(8):733–765
- Di Gasparo L, Roli A, Schaefer A (2007) EasyAnalyzer: an object-oriented framework for the experimental analysis of stochastic local search algorithms. In: Goos G, Hartmanis J, van Leeuwen J (eds) *Engineering stochastic local search algorithms. Designing, implementing and analyzing effective heuristics. Lecture notes in computer science*, vol 4638. Springer, Berlin-Heidelberg-New York, pp 76–90
- do Carmo R, de Campos G, de Souza J (2008) EasyMeta: a framework of metaheuristics for mono-objective optimization problems. In: *Anais do XL Simpósio Brasileiro de Pesquisa Operacional (SBPO2008)*. João Pessoa, Brazil
- Durillo J, Nebro A, Luna F, Dorronsoro B, Alba E (2006) jMetal: a Java framework for developing multi-objective optimization metaheuristics. *Tech. Rep. ITI-2006-10*, Departamento de Lenguajes y Ciencias de la Computación, University of Málaga
- Durillo J, Nebro A, Alba E (2010) The jMetal framework for multi-objective optimization: design and architecture. In: *Proceedings of the IEEE Congress on evolutionary computation (CEC) 2010*, pp 4138–4325
- Fink A, Voß S (2002) HotFrame: a heuristic optimization framework. In: *Optimization software class libraries*. Springer, Berlin-Heidelberg-New York, pp 81–154

- Fink A, Voß S, Woodruff D (1998) Building reusable software components for heuristic search. In: Kall P, Lüthi HJ (eds) Operations research proceedings 1998. Springer, Berlin-Heidelberg-New York, pp 210–219
- Fourer R, Ma J, Martin K (2008) Optimization services: a framework for distributed optimization. Tech Rep, COIN-OR
- Gagné C, Parizeau M (2006) Open BEAGLE: a C++ framework for your favorite evolutionary algorithm. *ACM SIGEVOlution* 1(1):12–15
- Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading
- Jünger M, Thienel S (2000) The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Softw Pract Exp* 30:1325–1349
- Johnson D (2002) A theoretician's guide to the experimental analysis of algorithms. Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges, pp 215–250
- Kleppe A, Warmer J, Bast W (2003) MDA explained: the model driven architecture: practice and promise. Addison-Wesley, Boston
- Lang JC (2008) Multi-location transshipment and substitution problems: an application to blood banks. Tech Rep, Schriften zur Quantitativen Betriebswirtschaftslehre, Department of Law, Business Administration and Economics, Technische Universität Darmstadt, Germany
- Lang JC, Domschke W (2010) Efficient reformulations for dynamic lot-sizing problems with product substitution. *OR Spectrum* 32(2):263–291
- Lang JC, Shen ZJM (2011) Fix-and-optimize heuristics for capacitated lot-sizing with sequence-dependent setups and substitutions. *Eur J Oper Res* 214(3):595–605
- Lang JC, Widjaja T, Buxmann P, Domschke W, Hess T (2008) Optimizing the supplier selection and service portfolio of a SOA service integrator. In: Proceedings of the 41st annual Hawaii international conference on system sciences, pp 89–98
- Law A (2006) Simulation modeling and analysis, 4th edn. McGraw-Hill, New York
- Lewis J, Neumann U (2003) Performance of Java versus C++. Tech Rep, Computer Graphics and Immersive Technology Lab, University of Southern California
- Linderoth J, Ralphs T (2004) Noncommercial software for mixed-integer linear programming. In: Karlof JK (ed) Integer programming: theory and practice. CRC Press, Boca Raton pp 253–304
- Lougee-Heimer R (2003) The common optimization interface for operations research: promoting open-source software in the operations research community. *IBM J Res Dev* 47(1):57–66
- Moret B (2002) Towards a discipline of experimental algorithmics. In: Goldwasser MH, Johnson DS, McGeoch CC (eds) Data structures, near neighbor searches, and methodology: 5th and 6th DIMACS implementation challenges: papers related to the DIMACS challenge on dictionaries and priority queues (1995–1996) and the DIMACS challenge on near neighbor searches (1998–1999). DIMACS series in discrete mathematics and theoretical computer science, vol 59. American Mathematical Society, pp 197–213
- Moret B, Shapiro H (2001) Algorithms and experiments: the new (and old) methodology. *J Univers Comput Sci* 7(5):434–446
- Ralphs T, Güzelsoy M (2006) The SYMPHONY callable library for mixed integer programming. The next wave in computing, optimization, and decision technologies, pp 61–76
- Saltzman M (2002) COIN-OR: an open-source library for optimization. In: Nielsen SS (ed) Programming languages and systems in computational economics and finance, chap 1. Kluwer, Boston, pp 3–32
- Spillner A, Linz T, Schaefer H (2007) Software testing foundations: a study guide for the certified tester exam, 3rd edn. Rocky Nook, Santa Barbara
- Stolte C, Tang D, Hanrahan P (2008) Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun ACM* 51(11):75–84
- Stonebraker M (2010) SQL databases v. NoSQL databases. *Commun ACM* 53(4):10–11
- Stützle T, Birattari M, Hoos H (2009) Engineering stochastic local search algorithms—designing, implementing and analyzing effective heuristics. In: Proceedings of second international workshop, SLS 2009, Brussels, Belgium, September 3–4, 2009. Lecture notes in computer science, vol 5752. Springer, Berlin-Heidelberg-New York
- Taillard E (2005) Few guidelines for analyzing methods. In: Proceedings of the 6th metaheuristics international conference (MIC 2005), Vienna, Austria
- Voß S, Woodruff D (2002) Optimization software class libraries. Kluwer, Boston