programming languages, e.g. see [2], [3] and [4, 5, 6] for, respectively, Java, C# and Python implementations.

This paper considers two design patterns: Template Method and Strategy, which were originally introduced in the GOF book and then, in 2010, indicated by Gamma, Helm and Johnson as two of the eight core patterns [7]. We show how these patterns can be used in a program which aims at solving different scheduling problems using the same metaheuristic algorithm. The benefits offered by these design patterns as well as their drawbacks are discussed. Implementation examples in the Python programming language are provided.

1. Metaheuristic

1.1 Exact algorithms, heuristics and metaheuristics

In the engineering practice, one often deals with optimization problems which are hard to solve (belong to the class of NP-hard problems). Such problems can be solved to optimality by exact algorithms only for small-sized instances. Generally, an exact algorithm provides an optimal (the best possible) solution in a finite amount of time, but unfortunately this finite time may increase exponentially with the problem size for the NP-hard problems.

Heuristics on the other hand, do not guarantee optimal solutions (they provide approximate solutions which are usually worse than the optimal ones) but they use a "reasonable" amount of time. Many heuristics are problem dependent, which means that their design is tightly connected with the solved problem, as they try to take advantage of the problem properties. They often use some greedy techniques and, during the search process, move only to solutions that immediately improve the current solution. Therefore, they can be easily trapped in local optima which are quite far from the optimal solution.

Metaheuristics are the "more general heuristics" which define frameworks that can be used for wide ranges of problems. However, the details of these frameworks, e.g. the ways in which solutions are represented and evaluated, remain always problem dependent and must be adapted to the solved problem. During the search process, metaheuristics may temporary move to solutions that are worse than the current

lowered. Thus, the probability of accepting worse solutions decreases and the algorithm is able to exploit the promising areas more thoroughly. When a new solution is “much worse” than the current one, the value of Δ is great, and such solution is accepted with low probability. When a new solution is only slightly worse than the previous one, it is accepted with a high probability, as Δ has a small value in such case. In Figure 1, we can see how the values of probability $P = e^{-\Delta/T}$ decrease with the increase in Δ/T.

Usually, at a fixed level of the temperature, a number of new solutions, L, are generated and checked. The temperature is decreased (every L iterations of the algorithm) by multiplying its current value by a reduction factor r ($0 < r < 1$). The search process continues while the temperature is greater than some minimal value, T_{min} (if $T \leq T_{min}$ the optimization process becomes “frozen”). The pseudocode of the SA algorithm is shown in Figure 2.

1. Generate an initial solution s and calculate its objective function value $f(s)$;
2. $s_{best} = s$; {remember the current solution s as the best found so far, s_{best}}
3. Set the values of the SA algorithm parameters: T, T_{min}, r, L;
4. **while** $T > T_{min}$ **do** {while an optimization process is not „frozen”}
5.   **for** $i = 1$ **to** L **do** {for a fixed value of the temperature T, repeat steps 6-13 L times}
6.   Generate a neighbor s' of solution s and evaluate it by calculating the objective function value $f(s')$;
7. $\Delta = f(s') - f(s)$;
8.   **if** $\Delta \leq 0$ **then** {if new solution s' is not worse than the current one, s}
9. $s = s'$; {accept the new solution as the current one}
10.   **if** $f(s) < f(s_{best})$ **then** $s_{best} = s$; {if the new solution is better than the best solution found so far, s_{best}, update s_{best}}
11.   **else** {if the new solution is worse than the current one}
12.   **if** $rand[0,1) < e^{-\Delta/T}$ **then** $s = s'$; {accept the new solution with probability $e^{-\Delta/T}$}
13.   **end if**;
14.   **end for**;
15. $T = rT$ {reduce the temperature}
16. **end while**;
17. **return** s_{best}; {return the best solution found}

Figure 2. Pseudocode of the SA algorithm

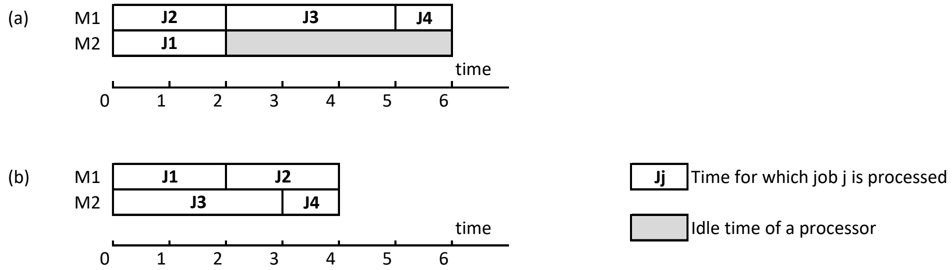

Figure 3. Schedules obtained for (a) $s = [2, 1, 1, 1]$ and (b) $s = [1, 1, 2, 2]$.

In problem P1, the objective is to minimize the maximum job completion time, C_{max} (i.e. the length of the schedule), which is given by the following formula:

$$C_{max} = \max_j C_j \quad (1)$$

where C_j is the completion time of job J_j.

In our example, $C_{max} = \max\{2, 2, 5, 6\} = 6$ for $s = [2, 1, 1, 1]$, and $C_{max} = \max\{2, 4, 3, 4\} = 4$ for $s = [1, 1, 2, 2]$. So, $s = [1, 1, 2, 2]$ represents a better schedule than that given by $s = [2, 1, 1, 1]$.

3.2 Problem P2

Problem P2 can be formulated as follows. There are n jobs to be performed by 2 dedicated machines. Each job has to be processed first on machine 1 and then on machine 2 (dedicated machines can represent specialized teams of workers or a client-server configuration in a production or computer environment). The time for which job J_j is processed on machine M_i is denoted by p_{ij}. The aim is to find an order in which jobs pass through the machines so as to minimize the mean flow time, denoted by $\bar{F}$ (i.e. the mean response time of a system).

Let us consider the following example.

Assume that $n = 3$ jobs are to be processed in a system consisting of $m = 2$ dedicated machines. The values of processing times of jobs are given in Table 2.

3. SA algorithm for problems P1 and P2

When using the SA algorithm to solve problems P1 and P2, some of the algorithm steps are performed in different ways depending on the characteristics of the solved problem, namely: the generation of an initial solution, creation of a neighbor solution and calculation of the objective function value. This is caused by the fact that solution representations and objectives are different for problems P1 and P2. For problem P1, the solution (an assignment of machines to jobs) is represented by a sequence of the machine indices. So, an initial solution can be created as a sequence of random numbers taken from range $[1,\ m]$ in which repetitions are allowed. The solution of problem P2 (an order in which jobs are processed) is represented by a sequence of job indices. So, as an initial solution we can take a sequence of random numbers from range $[1,\ n]$ without repetitions.

While solving problem P1, a neighbor solution can be created by randomly changing a machine index for a randomly chosen position (job) in the current solution. When dealing with problem P2, a new solution must ensure that a job index is not duplicated. Therefore, the construction of a neighbor may consist in moving a randomly chosen job to a different, randomly chosen position.

The objective function values are calculated according to formulas (1) and (2) for problems P1 and P2, respectively.

3.1 Ordinary design

The most straightforward approach for the implementation of the SA algorithm for problems P1 and P2 is to create two separate SA functions, one for P1 and the other for P2. However, this means the repetition of great parts of the SA algorithm code and thus, the violation of the "don't repeat yourself" (DRY) principle of software development [10].

A better approach consists in moving the code which depends on the solved problem to several separate functions and leaving the remaining code in one function which calls the problem specific operations using conditional statement `if...else` (see Figure 5). This design, however, has a serious drawback. When we want to ex-

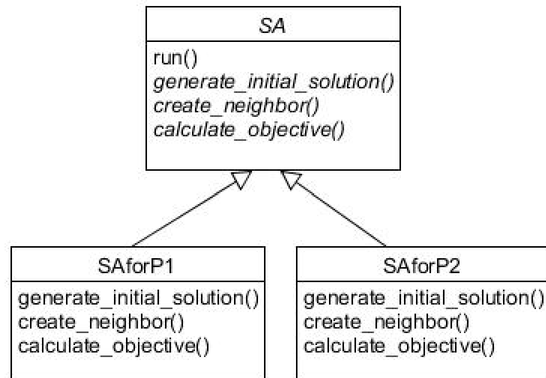

Figure 6. Template Method design pattern for the SA algorithm solving two scheduling problems

The Python implementation of the `SA` class with the `run()` method containing the algorithm skeleton is presented in Figure 7. The values for the algorithm parameters are set in a special method `__init__()` being a constructor. The `run()` method returns the best solution found by the algorithm along with its objective function value. Methods `random()` and `exp()` (used in line 18) are defined in the Python modules `random` and `math`, respectively.

```
1.    class SA:
2.        def __init__(self, T, Tmin, r, L):
3.            self.T, self.Tmin, self.r, self.L = T, Tmin, r, L
4.
5.        def run(self):
6.            s = self.generate_initial_solution()
7.            obj = self.calculate_objective(s)
8.            best_s, best_obj = s, obj
9.            while (self.T > self.Tmin):
10.               for i in range(self.L):
11.                   neighbor_s = self.create_neighbor(s)
12.                   neighbor_obj = self.calculate_objective(neighbor_s)
13.                   delta = neighbor_obj - obj
14.                   if delta <= 0:
15.                       s, obj = neighbor_s, neighbor_obj
16.                       if obj < best_obj:
17.                           best_s, best_obj = s, obj
18.                   elif random() < exp(-delta/self.T):
19.                       s, obj = neighbor_s, neighbor_obj
20.               self.T *= self.r
21.           return best_s, best_obj
```

Figure 7. Template Method design pattern: Python implementation of the `SA` class

```
sap1 = SAforP1()
s,o = sap1.run()
print('solution = {}, objective = {}'.format(s, o))


>>> solution = [0, 0, 1, 1], objective = 4
```

Figure 9. Template Method design pattern: running the algorithm

To prevent the instantiation of the `SA` class we can create it as an abstract class by deriving from `ABC` (abstract base class provided in Python module `abc`) and indicating methods `generate_initial_solution()`, `create_neighbor()` and `calculate_objective()` as abstract with `@abstractmethod` decorator (see Figure 10).

```
from abc import ABC, abstractmethod


class SA(ABC):
    ...

    @abstractmethod
    def generate_initial_solution(self):
        pass

    @abstractmethod
    def create_neighbor(self, s):
        pass

    @abstractmethod
    def calculate_objective(self, s):
        pass
```

Figure 10. Template Method design pattern: `SA` implemented as an abstract class

3.3 Strategy based design

According to [1] the strategy design pattern "defines a family of algorithms, encapsulates each one, and makes them interchangeable." A client dynamically chooses the algorithm that suits its current need.

In our program, we would like to dynamically choose the set of problem specific operations which are used for generating the initial solutions, creating the neighbors

```
class SA:

    def __init__(self, problem, T=10, Tmin=0.1, r=0.9, L=5):
        self.T, self.Tmin, self.r, self.L = T, Tmin, r, L
        self.problem = problem

    def run(self):
        s = self.problem.generate_initial_solution()
        obj = self.problem.calculate_objective(s)
        best_s, best_obj = s, obj
        ...


class P1:
    def __init__(self, n=4, m=2, p=[2, 2, 3, 1]):
        self.n, self.m, self.p = n, m, p
    ...


class P2:

    def __init__(self, n=3, p=[[4, 3, 1], [5, 1, 3]]):
        self.n, self.p = n, p
    ...
```

Figure 12. Strategy design pattern: fragments of the Python implementation of classes `SA`, `P1` and `P2`

```
p1 = P1()
sa = SA(p1)
s, o = sa.run()
print('solution = {}, objective = {}'.format(s, o))

p2 = P2()
sa = SA(p2)
s, o = sa.run()
print('solution = {}, objective = {}'.format(s, o))


>>> solution = [0, 0, 1, 1], objective = 4
>>> solution = [2, 1, 0], objective = 7.333333333333333
```

Figure 13. Strategy design pattern: running the algorithm

3.4 Benefits and drawbacks

Using the Template Method pattern or the Strategy pattern we avoid code duplication (there is one algorithm for both the problems) and code complication (no `if..else` statement is required for choosing a problem specific operation), and we create a program which complies with the Open-Closed Principle (“software entities should

be open for extension but closed for modification" [8]). Namely, to deal with a new problem, the program should be extended by a new subclass representing this problem (either a subclass of `SA` in case of the Template Method or a subclass of `P` in case of the Strategy based design). No modification of the existing code is needed. Moreover, a program applying the Strategy pattern can be extended by a new metaheuristic generating and evaluating neighbor solutions, e.g. a tabu search algorithm [11], without changing classes `P1` and `P2`.

So, the design of a program, in which a metaheuristic solves several optimization problems, can be considerably improved by using either the Template Method or the Strategy design pattern. However, these two patterns do not solve all the difficulties that may arise in such situation. A metaheuristic, to be successful with concrete real-life problems, usually requires some final tuning. For example, we may want to test different schemes for determining initial and neighbor solutions (such operations are often performed by some specialized algorithms). The considered design patterns do not provide much flexibility for such tuning task. It requires a separate class (subclass) for each combination of the tested schemes which may lead to a great number of subclasses and make testing and maintenance of the program quite difficult.

4. Final remarks

In this paper, we have shown in which ways the Template Method and Strategy patterns can improve the design of the program solving several different optimization problems by means of a metaheuristic algorithm. Several implementation examples written in the Python programming language have also been provided.

Literature

[1] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston 2003.

[2] Freeman E., Robson E., Sierra K., B. Bates, *Head First Design Patterns*, O'Reilly Media, Sebastopol-Boston 2009.

[3] Martin R.C., Martin M., *Agile Principles, Patterns and Practice in C#*, Prentice Hall, Upper Saddle River 2006.

[4] Kasampalis S., *Mastering Python Design Patterns*, Packt Publishing, Birmingham 2015.

[5] Phillips D., Giridhar Ch., Kasampalis S., Python: *Master the Art of Design Patterns*, Packt Publishing, Birmingham 2016.

[6] Summerfield, M., *Python in Practice*, Addison-Wesley, Boston, 2014.

[7] Gamma, E., Helm R., Johnson R., O'Brien L., *Design patterns 15 years later*, http://www.informit.com/articles/article.aspx?p=1404056 [Online access: November, 2017).

[8] Metropolis N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H., Teller E., *Equation of state calculations by fast computing machines*, "Journal of Chemical Physics" 21 (6), pp. 1087-1092, 1953.

[9] Błażewicz J., Cellary W., Słowiński R., Węglarz J., *Badania operacyjne dla informatyków*, Wydawnictwa Naukowo-Techniczne, Warszawa, 1983.

[10] Hunt A., D. Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, Boston, 1999.

[11] Glover, F., *Tabu search – part 1*, "ORSA Journal of Computing" 1, pp. 190-206, 1989.