Object Oriented Programming
École Centrale de Lille, G1/G2 – 2022-2023

Sequence 2 – Part 1 – Scheduling

Diego Cattaruzza, Pierre Hosteins, Maxime Ogier,
Frédéric Semet, Pablo Torrealba

Context and objectives

The French company Masks&Protection, based in the north of France, is currently facing a high demand for the production of single-use masks. This year, the demand is really exceptional and the production managers are not sure if there is enough time to produce all the masks. In addition, the contracts stipulated with the customers require that the company meets a given deadline for the delivery of the masks. If the deliveries are performed after the deadlines, the company ends up paying heavy penalties (proportional to the delay). It is therefore essential to respect as closely as possible the deadlines imposed by the customers.

Masks&Protection decided to hire the best students of the Hauts-de-France region to improve its production planning. The task is to design softwares able to provide a production planning capable of managing large quantities of demands.

The objective of this subject is to design a tool to help Masks&Protection in its work. First, we will focus on the data model to be implemented.

This subject will illustrate the following concepts:

- the notion of interface ;
- the management of collections of objects in Java ;
- the interface **List** and its implementations (**ArrayList** and **LinkedList**).

Good practices to adopt

- Test your code as you go along.
- Always respect the Java naming conventions.
- Make indentations in the right way.
- Give understandable and relevant names to your variables, attributes, classes, methods.
- Respect as much as possible the encapsulation principle.
- Use appropriate data structures.
- Make the methods short with few indentation levels.
- Do not duplicate pieces of code, but favor reusable code.
- Use the *refactoring* tools of your IDE (right-click then *Refactor* on Netbeans).
- Use Javadoc to understand how the methods you call work.
- When necessary, comment your own code according to the Javadoc format (comment starting with `"/**"` above the attribute and method definitions).

1 Modelling of the production workshop of Masks&Protection

The production workshop manager, Mister Safety, explains that his job is to plan the production of masks in the production workshop. This involves deciding which resources to use and the order in which the tasks are carried out to minimise the production time or costs, while respecting the constraints imposed by the production system.

Mister Safety explains that, in general, he has n identical machines in his production workshop. A task T_i is an elementary entity of work of processing time (duration) p_i located in time by a start time t_i or an completion time c_i ($c_i = t_i + p_i$). To fix ideas, a task corresponds to a set of identical masks to be delivered to the same customer. Moreover, in general, a task may have a delivery deadline d_i, and a unit penalty cost r_i to be paid for each minute of delay with respect to the deadline. Thud, the cost associated with a task will be $r_i \times \max\{c_i - d_i; 0\}$.

The construction of a schedule consists in assigning the tasks to the machines in order to establish the sequence of execution of the tasks on each machine (it is thus also necessary to decide the start time of each task). An example is given in Figure 1. In this example, we have 13 tasks. Task T_4 is assigned to machine M_1, it starts at $t = 3$ and ends at $t = 8$. At time $t = 8$, machine M_1 is available and task T_8 can start.

A task can be started on any machine as long as it is available. Thus, the start time of a new task on that machine is given by the completion time of the previous task on that same machine. There is no point in having waiting times. Moreover, Mister Safety explains that a task cannot be interrupted to be continued later.

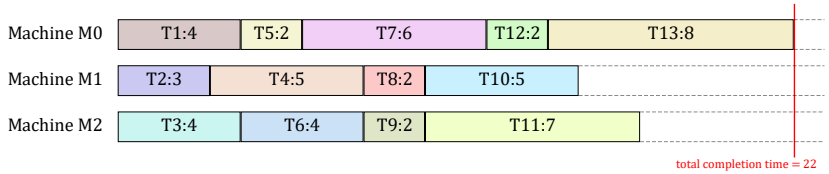

Figure 1: Example of a scheduling on 3 machines. Each task is denoted $T_i : p_i$.

1.1 Creation of the class Task

First, we are interested in the representation of a production task.

For this purpose, Mister Safety explains that in the production shop of Masks&Protection the execution of a task takes at least 50 minutes. When scheduling tasks, he considers time 0 as the initial time of the scheduling, and therefore the delivery times are given in minutes starting from time instant 0. It happens that some tasks have no delivery date: in this case, the delivery date is considered infinite and the penalty cost is zero. On the other hand, Mister Safety has managed to negotiate with the sales department that the delivery date of a task should be at least twice the execution time of the task (so $d_i \geq 2p_i$). Mister Safety says that the processing time cannot be changed and, being a good negotiator, he has also obtained from the sales department that the delivery dates and penalty costs cannot be changed. In addition, each task must be characterised by a unique identifier.

Question 1. After opening Netbeans, create a new project **Scheduling**, a package **workshop** (which will contain the production shop modelling classes), and write a class **Task** which represents a task to be performed by Masks&Protection. Define the attributes of this class **Task**. Comment each attribute in Javadoc format.

Question 2. Add a default constructor, a data constructor for execution time, and a data constructor for execution time, delivery date, and delay penalty. Don't forget that the Netbeans wizards are there to help you (right click and then *Insert Code...*). Comment each of these constructors in Javadoc format.

Question 3. Complete the **Task** class with the necessary getters and setters and a **toString()** method which returns a string containing information about the task (this is in fact a redefinition of the **toString()** method of the **Object** class). Don't forget that the Netbeans wizards are there to help you (right click then *Insert Code...*). Comment the methods you have just written in Javadoc format.

Question 4. Write a main method (**public static void main(String[] args)**) to test the use of the **Task** class. Be sure to test all possible cases and check that the production time and deadline recommendations are still met. Don't forget that the Netbeans wizards are there to help you (type "*psvm*" then *Ctrl* + *space*, or "*sout*" then *Ctrl* + *space*)!

1.2 Creation of the class Machine

1.2.1 Concept of interface

We saw in the previous subject that an abstract class allows to define functionalities in a base class which are common to all its descendants, while requiring to redefine certain methods. If we consider an abstract class which implements no method and has no attribute (except for the constants declared **static** and **final**), we end up with the notion of interface. The definition of an interface is presented like the one of a class. We simply use the keyword **interface** instead of **class**, as below:

```
1  public interface I {
2     void f(int n); // Header of a method f
3     double g() ; // Header of a method g
4  }
```

By definition, the methods of an interface are abstract and public, but it is not necessary to mention this with the keywords **abstract** and **public**. When defining a class, one can specify that it implements a given interface by using the keyword **implements** as in the following:

```
1  public class A implements I {
2     // A must (re)define the methods f and g of the interface I
3     @Override
4     public void f (int a) {
5       // Implementation of the method f
6     }
7
8     @Override
9     public double g () {
10      // Implementation of the method g
11    }
12 }
```

A class can inherit from a single class, but a class can implement multiple interfaces. This is how multiple inheritance can be handled in Java.

Polymorphism in object-oriented programming is a concept that allows an object to be manipulated without knowing its type (completely). In the previous subject, cars and people were manipulated in a Formula 1 team without knowing

exactly whether they were Formula 1 cars or trucks, drivers or technicians. An object was recognised as an instance of its class and all its parent classes. This notion of polymorphism is also valid for interfaces, i.e. an object is recognised as an instance of each of the interfaces it implements.

1.2.2 Collections of objects

To manipulate collections of objects in Java, the **java.util** package provides a set of classes that allow the manipulation of dynamic vectors, sets, linked lists, queues and associative tables. The classes related to vectors, lists, sets and queues implement the same interface (**Collection**) which they complete with their own functionalities.

Indeed, collections are handled by generic classes implementing the interface **Collection<E>**, where **E** represents the type of the elements of the collection. All the elements of a collection are thus of the same type **E** (or of a class derived from **E**).

1.2.3 Loop through the elements of a collection

The *for...each* loop allows you to loop through a collection of objects of type **E** named **col** in a simple way as follows:

```
1  for (E elem : col) {
2     // Use of elem
3  }
```

The variable **elem** successively takes the value of each of the references of the elements in the collection. **HOWEVER**, this scheme cannot be used if you wish to modify the collection using methods such as **remove()** or **add()**!

There are also specific objects, called "iterators", which allow you to loop through the elements of a collection one by one. They look like pointers (like those in C or C++) without having exactly the same properties. Each collection class has a method named **iterator** providing a one-way iterator, i.e. an object of a class implementing the **Iterator<E>** interface. We can therefore also loop through a collection of objects of type **E** named **col** in the following way:

```
1  Iterator<E> iter = col.iterator();
2  while (iter.hasNext()) {
3     E elem = iter.next();
4     // Use of elem
5  }
```

Indeed, the **iterator()** method returns an object pointing the first element of the collection, if it exists. Method **next()** provides the element pointed by **iter** and advances the iterator to the next position. If you wish to loop through the same collection several times, you can simply reset the iterator by calling the **iterator()** method again.

The interface **Iterator** provides method **remove()** which removes from the collection the last object returned by **next()**.

1.2.4 List of objects

A list of objects is an ordered collection of objects (each object has a position in the list). Several objects of the same value can therefore appear in a list, and the add and delete methods require as a parameter the position at which the operation should be performed.

Lists of objects in Java are objects that implement the interface **List** (which itself inherits from the interface **Collection**). **List** is an interface and so you cannot instantiate it. The Java Collection API provides two different implementations of the **List** interface, namely **ArrayList** and **LinkedList**: you can choose the one that suits you best. For the methods offered by these objects, see the Javadoc.

ArrayList is implemented as a dynamic array (i.e. an array of variable size). This provides fast access features comparable to those of an array of objects (in $\mathcal{O}(1)$). In addition, this class offers more flexibility, indeed, its size (number of elements) can vary during the execution of the code. However, the direct access to an element of a given rank is possible only if the locations of the objects (rather their references) are contiguous in memory (like those of an array). In addition, this class suffers from a flaw arising from the nature of the class itself: addition or deletion of an object at a given position will no longer be done in $\mathcal{O}(1)$ as in the case of a linked list, but only on average in $\mathcal{O}(n)$. In the end, the **ArrayList** are well suited to direct access, provided that the addition and deletion operations remain limited.

The **LinkedList** is implemented as a double-chained list. The great advantage of such a structure is that it allows additions or deletions at a given position with $\mathcal{O}(1)$ efficiency (this is achieved by a simple set of reference modifications). However, accessing an element according to its value or its position in the list will not be very efficient because it will inevitably require a loop through a part of the list. The efficiency will therefore be on average $\mathcal{O}(n)$. In the end, the **LinkedList** are well suited for the addition and deletion operations, provided that direct access to an element remains limited.

In general, there is no implementation of the **List** interface that is always better than another. The choice must be made according to the particular characteristics of your problem. This observation applies not only to implementations of the **List** interface, but also to other interfaces available in the Collection API (**Set**, **Map**).

1.2.5 Back at Masks&Protection

Masks&Protection has a limited number of machines to perform the tasks. Mister Safety who has been planning tasks for twenty years with his pen and paper explains that when assigning tasks to one of the machines, it is very important to know:

- the current availability date of the machine, i.e. the time instant from which a new task can be launched on the machine ;
- the cumulative penalty costs associated with this machine.

Every Friday evening, Mister Safety finds himself solving a very difficult problem: how to assign tasks to machines in order to minimise the total execution time of the schedule (i.e. the date at which all tasks are completed), and then if possible to minimise the penalty costs for delays.

We will try to help Mister Safety later. For now, Mister Safety explains that to perform his scheduling: initially he assigns each machine with a *task list* and, then, he progressively adds tasks to each of these lists. On each machine, the tasks are executed in the order in which they appear in the list. Figure 1 shows the complete UML diagram for this project.

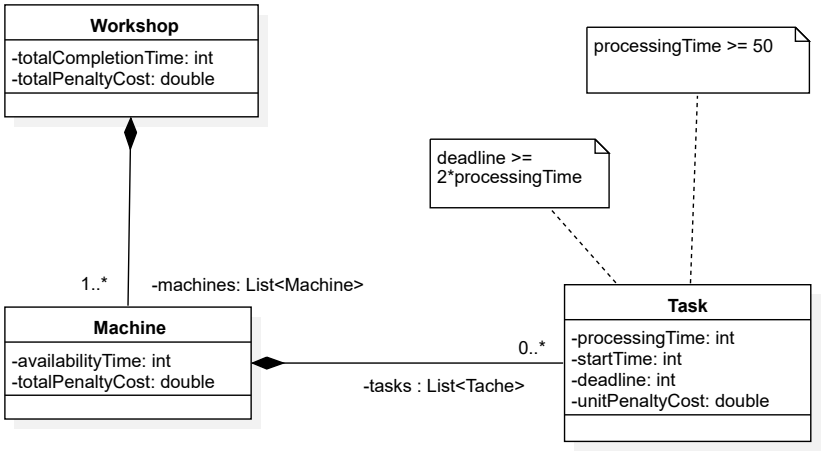

Figure 2: UML diagram.

Question 5. In the package **workshop**, add a class **Machine** with its attributes. It represents a machine in the workshop. Comment the code in Javadoc format.

Question 6. Add a default constructor to the class **Machine**. Add the necessary getters and setters.

Question 7. Add a method **public boolean addTask(Task t)** which adds task **t** to the list of tasks to be executed on the machine. The method returns **true** if the addition has been done correctly, **false** otherwise. It is possible to call the method **add(Object o)** of the interface **List** which allows to add an object to the list (the addition is done at the end). As a reminder, Mister Safety proposes that when a task is added to the list, it is executed as soon as the machine is available, in order to avoid dead times. Your method **addTask(Task t)** must also correctly update the machine and task data.

Question 8. Add a method **toString()** that returns a string containing the machine information (in a easily readable manner) and write a main method to test class **Machine**. What is the total processing time and penalty to be paid by Masks&Protection if it receives the following tasks (defined by processing time, delivery deadline and unit penalty cost) and runs them in order on a single machine? What are the start time and penalty cost for task **t5**?

```
1  Task t1 = new Task(150, 300, 2.5);
2  Task t2 = new Task(140, 400, 1.5);
3  Task t3 = new Task(50, 200, 2.5);
4  Task t4 = new Task(85, 200, 1.0);
5  Task t5 = new Task(75, 160, 0.5);
6  Task t6 = new Task(80, 500, 1.5);
```

1.3 Creation of the class Workshop

In the first part of this subject, you got to know Mister Safety. In the course of your conversations, you have shown him your intelligence and insight. So he decides to open you the doors of the Masks&Protection's workshop.

The workshop mainly consists of a limited number of machines on which the tasks are performed. In addition, Mister Safety explains that he needs to know the total completion time (the time at which all the tasks are completed) and the total penalty cost (the sum of the penalty costs of the machines).

Question 9. In the **workshop** package, add a **Workshop** class with its attributes. The workshop represents the core of Masks&Protection. The set of machines is stored in a list container. What type of list do you choose? To make the right decision, consider that you will need to call the list interface's **get()** method as many times as the number of tasks (in order to select the machine to run the task on). Check whether the **get()** method is faster in the **ArrayList** or **LinkedList** implementation. Important sources of information are the Javadoc and the website `http://docs.oracle.com/`. Comment the code in Javadoc format.

Question 10. Write a constructor which takes as a parameter the number of machines in the workshop (it is assumed that there is at least one machine, otherwise it is not a real workshop). Add the necessary getters and setters, as well as a method **toString()** which returns a string containing the information about the workshop. Comment your code and write a main method to test the **Workshop** class.

Question 11. Add a method **public void updateCriteria()** which calculates the scheduling criteria (total completion time and total penalty cost) and updates the corresponding attributes.

Mister Safety explains that on Friday evening, before leaving for the weekend, he designs the following week's schedule with this modus operandi:

- The sales manager gives him a list of all the tasks to be completed during the week;
- Mister Safety considers the tasks one by one in the order they are listed;
- The first task of the list is assigned to the first machine;
- For each of the remaining tasks, Mister Safety assigns it to the machine that becomes available first (the one with the smallest availability date).

In addition, Mister Safety specifies that the machines can work continuously without interruption.

Question 12. Add to your class **Workshop** the necessary methods to implement the modus operandi proposed by Mister Safety:

- a method **private Machine getMachine(int posMachine)** which returns the machine in position **posMachine** of your list of machines (you may call the method **get()** of the interface **List**) ;
- a method **private boolean addTask(Task t, int posMachine)** which adds task **t** to the machine in position **posMachine** in your list of machines (you may call the method **addTask()** of the class **Machine**) ; this method returns **true** if the addition was correctly performed, **false** otherwise ;
- a method **private int getFirstAvailableMachine()** which returns an integer indicating the position in the list of machines of the first available machine;
- a method **public void scheduleTasks(List<Tasks> tasks)** which schedules the tasks of list **tasks** according to the modus operandi of Masks&Protection, then updates the scheduling criteria (total completion time and total penalty cost).

Comment these methods in Javadoc format.

Question 13. Test the scheduling method for a two-machine workshop on the set of tasks provided in Question 8. What is the total completion time? What is the total penalty cost?

Question 14. If we are interested in minimising the total completion time of the tasks on the machines, what do you think about the modus operandi proposed by Mister Safety? Based on the test performed in the previous question, can you propose a task scheduling that allows to decrease the total completion time? What value do you obtain? Propose a task scheduling algorithm that minimises the total execution time.

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*