isep Instituto Superior de Engenharia do Porto

Production Schedule

Técnicas Avançadas de Programação

Mestrado Engenharia Informática

Filipe Ferreira - 1160826 | Rute Santos – 1160663 | Vera Dias – 1160941

Abstract

This document is developed within the scope of the curricular unit Técnicas Avançadas de Programação (TAP) of Mestrado em Engenharia Informática (MEI).

The project aims to design and development of an application using functional programming techniques. The problem to solve centers around the scheduling of orders in a factory.

The development of the application was divided into three milestones applying the different concepts of functional programming in Scala.

Keywords: Algorithm, Function Programming, Property Based Test, Scala, Schedule

Table of Contents

1 Introduction

This document describes the elaboration of a curriculum project for the creation of a production line schedule.
This chapter intends to present the context associated with the elaboration of this project, briefly describe the problem that this project needs to solve and, finally, describe the structure of the document.

1.1 Context

The present project was developed in the scope of the curricular unit Técnicas Avançadas de Programação (TAP) of Mestrado em Engenharia Informática (MEI) at Instituto Superior de Engenharia do Porto (ISEP). This project aims to address the scheduling of production orders in a factory and is divided into three distinct milestones.

1.2 Problem Description and Objectives

The project must be developed in the functional programming language Scala.
Milestone 1 intends to develop a *first in, first out* (FIFO) algorithm for scheduling the necessary tasks of the products of a set of orders.
Milestone 2 intends to develop tests based on Domain Properties (PBT).
Finally, *Milestone 3* aims to optimize the scheduling of product tasks to maximize the use of resources, reducing the total time of the production line.
All milestones must have tests to validate the correct functioning of the developed algorithms.
Input and output files are provided to clarify the objective of the work that the developed algorithms must perform.

1.3 Document Structure

This document, in addition to the introduction, where the development of the entire project is contextualized, begins with the presentation of the problem domain. Next, all the work carried out in *Milestone 1* for the development of a Schedule with the FIFO algorithm is described. Later, *Milestone 2* relating to *Property Based* tests is presented, and as the last milestone, *Milestone 3* is described, where an algorithm was made to optimize the use of resources.
Finally, the necessary conclusions are drawn from all the work developed.

2 Production Schedule Domain

This section presents the domain for the Production Schedule. It describes all the domain concepts and the different inputs and outputs generated from the algorithm.

2.1 Domain Concepts

According to the document given for milestone one, a factory produces different orders. Each order has several products to be produced. A production is a combination of a sequence of tasks, to complete a task it's necessary to use a subset of physical resources for a given duration. To operate a physical resource, a human resource must be used. Each physical resource and human resource have one or more resource types. Given a list of orders and resources, the main goal of this project is to generate a schedule of production. A schedule of production begins at time 0 and ends when the last task finishes.

Table 1 translates all the domain concepts described before.

Class	**Description**
HumanResource	Entity that contains all the information for a Human Resource, it is identified by a *String* Id, name of the Human Resource and a list of *skills* that will be used to establish a connection with *PhysicalResource* that will be used in the entity *Task*.
Order	This is one of the main input concepts in this domain. Identified by a *String* type Id and references a product and the required production quantity. Through its product, it is possible to reach all the entities necessary for the fulfillment of the *Order*.
PhysicalResource	Identified by an Id of the type *String*, this entity contains the *ResourceType* that will be used to establish a connection with HumanResource with the ability to operate it and the need to use it in a *Task*.
Product	The products contain production processes, which consist of the *Tasks* with *ResourceType* necessary for it to be successfully built. An *id* is required to identify the product in a *Order* and a specific name is required.
ResourceType	This entity is transversal to several other entities in the domain (*HumanResource*, *PhysicalResource*, *Production*, *Task*) and consists of the types of resources that the system supports.
Task	This entity is the related to a *Product* since a Product contains a list of tasks necessary to be created. A *Task* has an Id of type *String*, a time in unit time and a list of *Resource Types*.

Table 1 - Different domain concepts.

All these domain concepts interact in the project and during the implementation of the algorithm. Before the algorithm starts it is necessary to import the objects from the *XML* file. After everything is compiled, the object *Production* is produced and this object aggregates and has the knowledge about every domain concept such as *Orders*, *Products*, *Tasks*, *Physical Resources*, *Human Resources*, and *Resource Types*. All the interactions can be reviewed in the following domain model, represented by Figure 1.

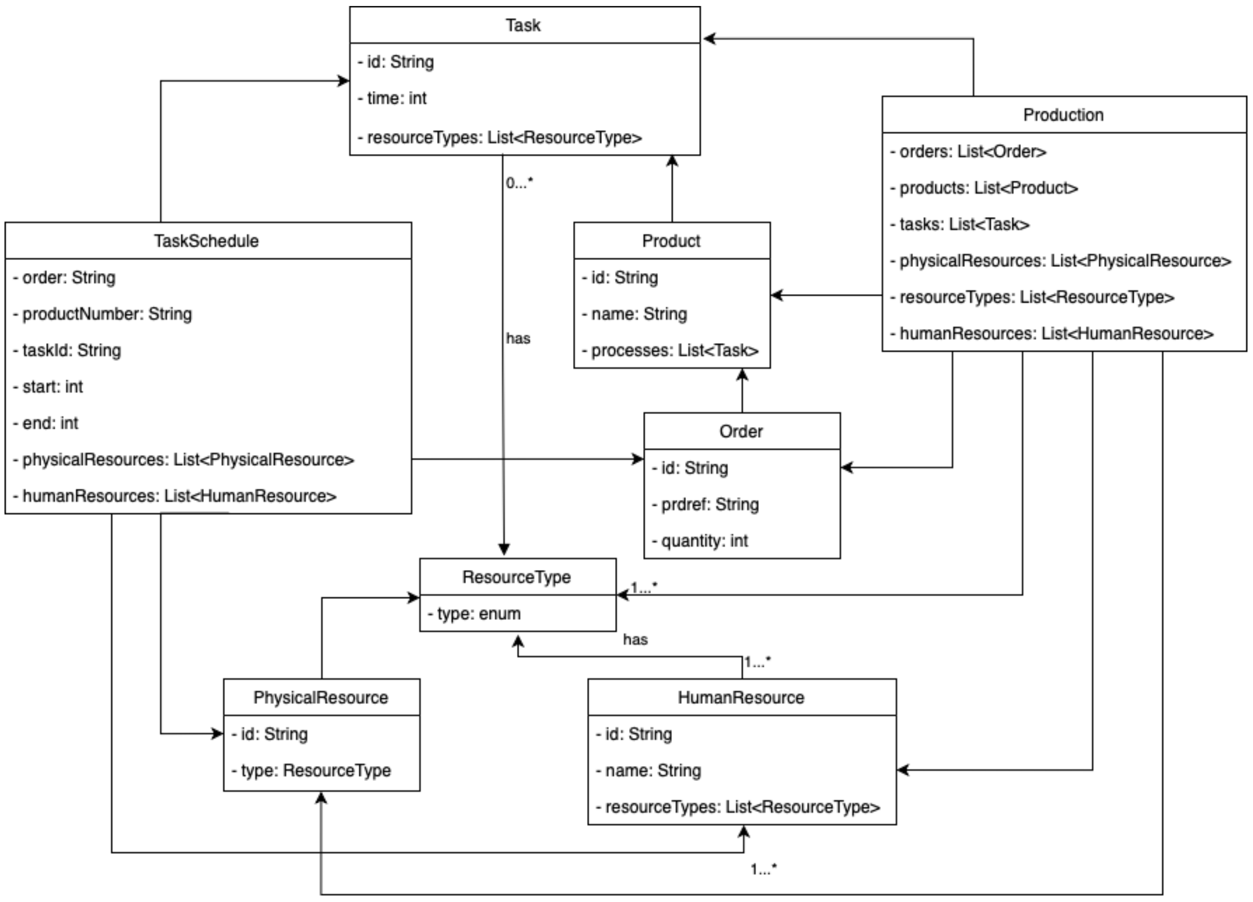


Figure 1 - Domain model diagram.

2.2 Input and Output Expected

As mentioned above, the input and the output file generated by the algorithm must follow a specific guideline. The input file should contain the list of resource types available for that production, a list of products a list of human resources, and a list of orders. An example of a snippet of an input file is shown in Figure 2.

```
  <PhysicalResources>
      <Physical id="PRS_1" type="PRST 1"/>
      <Physical id="PRS_2" type="PRST 2"/>
  </PhysicalResources>
  <Tasks>
      <Task id="TSK_1" time="100">
        <PhysicalResource type="PRST 1"/>
        <PhysicalResource type="PRST 2"/>
      </Task>
  </Tasks>
  <HumanResources>
      <Human id="HRS_1" name="Antonio">
        <Handles type="PRST 1"/>
        <Handles type="PRST 2"/>
      </Human>
  </HumanResources>
  <Products>
     <Product id="PRD_1" name="Product 1">
       <Process tskref="TSK_1"/>
       <Process tskref="TSK_2"/>
     </Product>
  </Products>
  <Orders>
     <Order id="ORD_1" prdref="PRD_1" quantity="1"/>
  </Orders>
```

Figure 2 - Example of a simple input file.

Lastly, the generated output file is also important to be mentioned. When the algorithm generates a list of Task Schedules, each element contains the information about the reference order, the number of the product, the reference of the task, the beginning start and ending time of this task, and finally the list of physical resources and human resources for that task. A simple representation of the output file can be represented in Figure 3.

```
<TaskSchedule order="ORD_1" productNumber="1" task="TSK_1" start="0" end="100">

      <PhysicalResources>

            <Physical id="PRS_1" />

            <Physical id="PRS_4" />

      </PhysicalResources>

      <HumanResources>

            <Human name="Antonio" />

            <Human name="Maria" />

      </HumanResources>

</TaskSchedule>
```

Figure 3 - Example of a Task Schedule.

3 Milestone 1 – First In, First Out Scheduling Algorithm

This section presents the steps for the achievement of Milestone 1. It describes the import of the input files, the development of the first version of the algorithm, and finally, all the tests performed to validate it.

3.1 Import and Parse Input Files

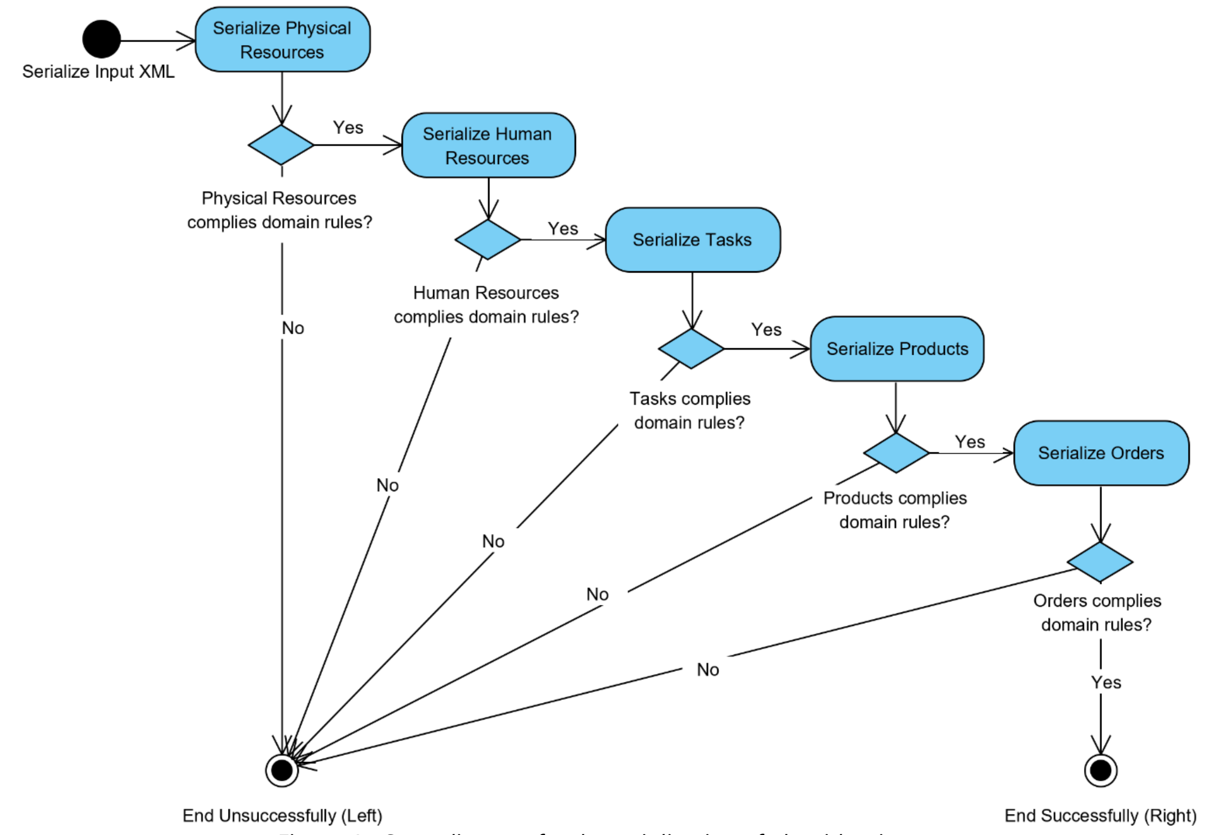
To schedule the production of the product tasks, its necessary to import the *XML* files supplied, with all domain concepts information. This information will be validated and processed by an algorithm to create the output *XML* files, with all the schedules.
The Table 2 shows the validations made to the imported data.

Object Class	**Validation**
HumanResource	ID must follow the pattern *HRS(_|)[0-9]+* Name must not be null or empty
Order	ID must follow the pattern *ORD(_|)[0-9]+* Name must not be null or empty
PhysicalResource	ID must follow the pattern *PRS(_|)[0-9]+*
Product	ID must follow the pattern *PRD(_|)[0-9]+* Name must not be null or empty
ResourceType	Must follow the pattern *PRST(_|)[0-9]+*
Task	ID must follow the pattern *TSK(_|)[0-9]+* Name must not be null or empty

Table 2 - Domain validations using regular expressions.

The input *XML* is used to create a *Production* object from an *XML Node*. To create a *Production* object, it is also necessary to create everything else. To do this the method *traverse* is used for each attribute of the *Production* element. To create a *Production* element is necessary to parse the *Physical Resources*, if this does not comply with the domain rules (described in Table 2) then the import is unsuccessful and is generated an error in the file.
In the case that the *Physical Resource* is valid then it is necessary to parse the *Human Resources* that will follow a similar flow described above. After this, the *Tasks* are imported (this will fail if the tasks do not comply with the domain rules), after this the *Products* are parsed and after that the *Orders*. Every object should respect the domain rules so that the end is successful, and the algorithm can generate a *Task Schedule*.
This flow should be sequential since an *Order* needs a *Product*, a *Product* needs a list of *Tasks*, and this has a list of *Resource Types* that are used by *Human Resources* and *Physical Resources*.

The diagram presented in Figure 4 represents the flow of serialization before the algorithm begins.



Serialize Input XML
Serialize Physical Resources
Physical Resources complies domain rules?
Yes
No
Serialize Human Resources
Human Resources complies domain rules?
Yes
No
Serialize Tasks
Tasks complies domain rules?
Yes
No
Serialize Products
Products complies domain rules?
Yes
No
Serialize Orders
Orders complies domain rules?
Yes
No
End Unsuccessfully (Left)
End Successfully (Right)

Figure 4 - State diagram for the serialization of algorithm input.

As it is possible to verify in Figure 4, if the parse of any domain concept fails, the execution of that stage will stop, and an *XML* file with the *DomainError* will be generated and exported as result.

3.2 Export Ouput files

The result of the algorithm should be exported as an XML element with each element of the *Task Schedule* produced.

Each element is serialized using an auxiliary method in the *XMLParser* class. For each *Task Scheduler* element, there is an *order id*, *product number*, *task id*, *start* and *end*, a list of *Physical Resources*, and a list of *Human Resources*. For each serialization, the *scala.xml* library is used to create a flexible *XML* node, the following approach was used:

```
<A attribute={ value1 }>
  { value2 }
</A>;
```

Each serialization returns an Elem that after all the serialization finishes the Elem is written in a file using the method *save()* from the auxiliary class *XML* in the project. If an error occurs along the whole process the serialization does not occur, and a *DomainError* is returned instead of an Elem.

3.3 FIFO Algorithm

The *First In, First Out* algorithm consists in, given an information structure, the first element being processed is the first element returned. For example, there is a ticket counter where people come, take a ticket, and go away, this people came in line, so the first person enter in the queue will get the ticket first and leave the queue, the person entering in the queue next will get the ticket after the first person and consequently leave the queue after [1]. That is the same logic applied to the task schedule in this algorithm.

The figure bellow presents the organization of package *Schedule*:

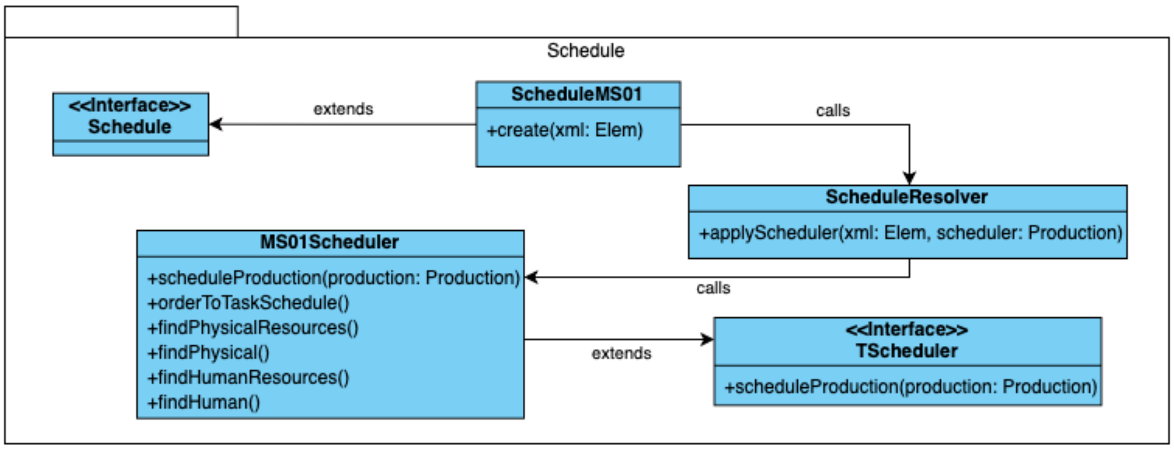


Figure 5 - Schedule package model.

As a good practice, for the development of the algorithm, the class where the algorithm is implemented (*MS01Scheduler*) extends an interface (*TScheduler*) that will be called to execute the logic. The method *scheduleProduction* is called by the *ScheduleResolver* and this class is used in *ScheduleMS01*. In this waynwe can abstract the addition of other algorithms with their class.

The algorithm for the creation of *Schedule* uses *tail recursion* and consists of the following steps:

- Iterate each *Order* that *Production* contains;
- Call recursively a method to create a TaskSchedule as many times as the *productNumber* it contains;
- This recursive call will stop when the *productNumber* is reached and does the sum required to *startTime* and *endTime*;
- During each recursive iteration, the necessary and available *Physical* and *Human* for the task is obtained;
- The final output is a list of type *TaskSchedule*.

In order to facilitate the interpretation of how the developed algorithm works, the sequence diagram illustrated in Figure 6 describes the phases of execution of the algorithm:

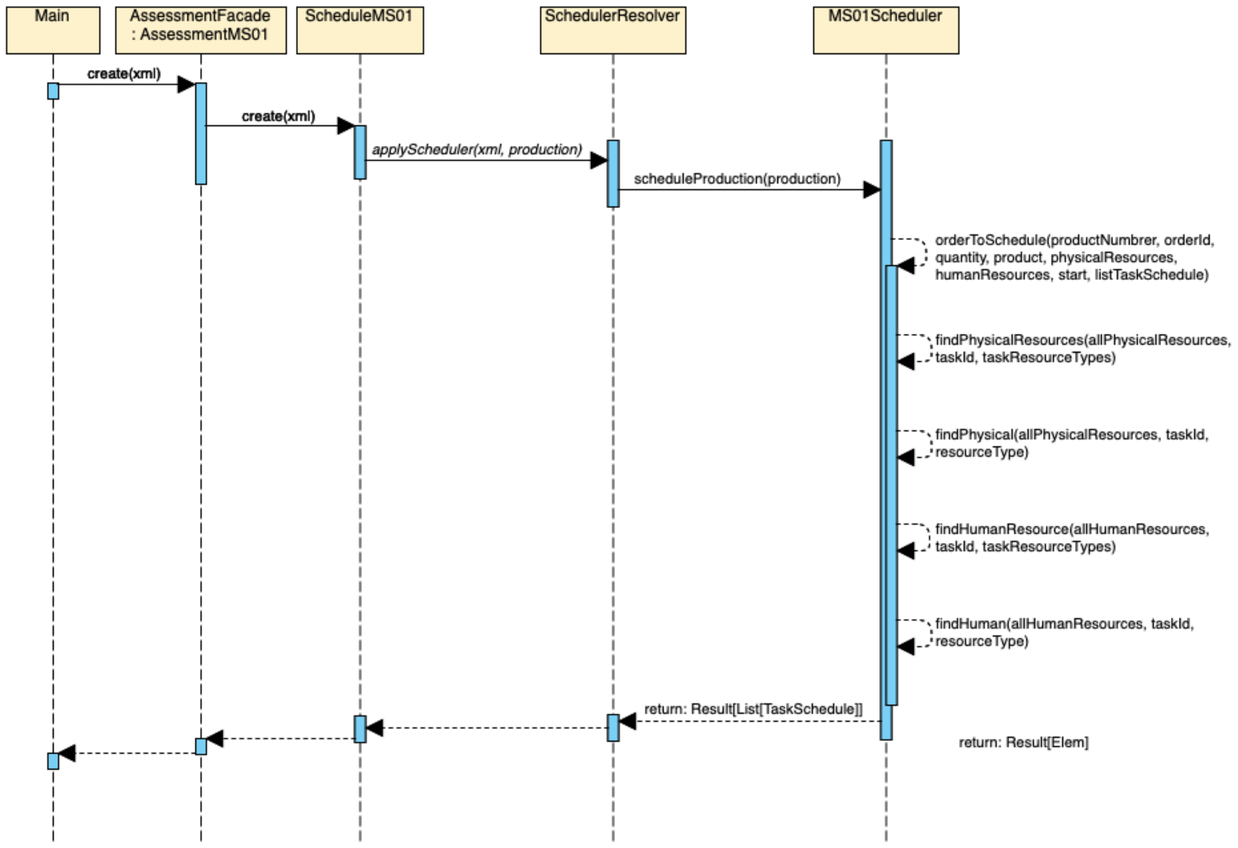

Main
AssessmentFacade : AssessmentMS01
ScheduleMS01
SchedulerResolver
MS01Scheduler
create(xml)
create(xml)
applyScheduler(xml, production)
scheduleProduction(production)
orderToSchedule(productNumbrer, orderId, quantity, product, physicalResources, humanResources, start, listTaskSchedule)
findPhysicalResources(allPhysicalResources, taskId, taskResourceTypes)
findPhysical(allPhysicalResources, taskId, resourceType)
findHumanResource(allHumanResources, taskId, taskResourceTypes)
findHuman(allHumanResources, taskId, resourceType)
return: Result[List[TaskSchedule]]
return: Result[Elem]

Figure 6 - Sequence diagram of Milestone 1 Algorithm.

As a result of the execution of the algorithm we have the Schedule in the format presented in the previous section.

3.4 Tests

Unit and functional tests were created to assess the correct functionality of the algorithm implemented. The unit testing framework used was *scalatest*.

To verify the validations described in Table 2 multiple unit test cases were created to test the format of different positive and negative cases. The class *SimpleTypesTests* contains the tests implemented to test the various Simple Types created.

Additionally, it was created for each domain concept the respective test class to verify the smart constructors, for example, for concept Product it was created the class *ProductTests*.

Regarding the functional tests, the classes *MS01SchedulerTests* and *ScheduleMS01Tests* were created to validate the import and parse of the import, the algorithm as well as the creation of the *XML* output.

Furthermore, it was also used the files provided to accurately test the algorithm.

4 Milestone 2 – Property Based Tests

This section presents the steps for the achievement of Milestone 2. It begins by describing the data generators developed and then as domain properties will be used as tests.

4.1 Generators

Before implementing any Property Based Test it is necessary to implement Generators. This helps create different elements with determined values.
A list of the used Generators is enumerated in Table 3:

Generator	**Description**
idGenerator[A](prefix: String, getId: String => Result[A]): Gen[A]	Generates an unique identifier with a specified prefix
idGenerator[A](prefix: String, number: Int, getId: String => Result[A]): Gen[A]	Generates an unique identifier with a specified prefix of Type A using function getId to convert String into A
nameGenerator[A](getName: String => Result[A]): Gen[A]	Generates a random string and convert to a *SimpleType*
filteredListGenerator[A](list: List[A]): Gen[List[A]]	Pick some elements from a generated list
listGenerator[A](generator: Int => Gen[A]): Gen[List[A]]	Generates a list of elements as well as its size
listGenerator[A](size: Int, generator: Int => Gen[A]): Gen[List[A]]	Recursive generator that receives size of list to generate
physicalResourceGenerator(id: Int): Gen[PhysicalResource]	Generates a valid physical resource with a received ID
listPhysicalResourceGenerator: Gen[List[PhysicalResource]]	Generates a list of valid physical resources without repeated IDs
humanResourceGenerator(resourceTypesList: List[ResourceType])(id: Int): Gen[HumanResource]	Generates a valid human resources with a given ID
listHumanResourceGenerator(resourceTypesList: List[ResourceType]): Gen[List[HumanResource]]	Generates a list of valid human resources without repeated IDs

taskGenerator(resourceTypesList: List[ResourceType])(id: Int): Gen[Task]	Generates a valid task referring valid resources, with a received unique identifier
listTaskGenerator(resourceTypesList: List[ResourceType]): Gen[List[Task]]	Generates a valid list of task without repeated IDs
productGenerator(taskList: List[Task])(id: Int): Gen[Product]	Generates a valid product with a given ID referring valid tasks
listProductGenerator(taskList: List[Task]): Gen[List[Product]]	Generates a valid list of product without repeated IDs
orderGenerator(productList: List[Product])(id: Int): Gen[Order]	Generates a valid order with a unique identifier, referring to a valid product
listOrderGenerator(productList: List[Product]): Gen[List[Order]]	Generates a valid list of order without repeated IDs
productionGenerator: Gen[Production]	Generates a valid production
productionGeneratorPhysicalResourceUnavailable: Gen[(Production, TaskId, ResourceType)]	Generates an invalid production where there are not enough PhysicalResources to complete a task of an order
productionGeneratorHumanResourceUnavailable: Gen[(Production, TaskId, ResourceType)]	Generates an invalid production where there are not enough HumanResources to complete a task of an order

Table 3 - Used Generators.

4.2 Property Based Tests

In the previous milestone, it was included unit tests to validate the model of the domain and program accuracy. These types of tests are called example-based testing and do not consider all possible domain values (function inputs), and it is possible that we may fail to anticipate edge cases that cause errors in the application.
Property Based Testing (PBT) does not need concrete examples of what is expected as inputs of the function under test. PBT stretches the boundaries of the inputs to the limit, possibly uncovering failing behavior. To conclude, properties are general rules that describe a program's behavior, whatever the input is, the defined property condition must be always true [2].

Based on the information about Property Based Testing, the following tests were implemented as represented by the following.

- The same resource cannot be used at the same time by two tasks.
- The complete schedule must schedule all the tasks of all the *Products* needed.
- The number of *TaskSchedules* must be the same as the result of the multiplication between the *Tasks* and the *quantity* of the *Product* in an *Order*.
- The end time of the last *TaskSchedule* must be equal to the multiplication between the sum time of the *Tasks* and the *quantity* of the *Product* in each *Order*.
- The schedule should not be implemented when there are not enough resources (*PhysicalResource* and *HumanResource*) to process the Task Schedule.
- The sum of the *TaskSchedule* time of an order should be equal to sum of the times of the *Task* multiplied by the *quantity*.
- The *TaskSchedules* cannot contain *PhysicalResource* that is not valid.
- The *TaskSchedules* cannot contain *HumanResource* that is not valid.
- The highest *productNumber* for each *Order* should be equal to the *quantity* to that *Order*.
- For a given *TaskSchedule*, the list of *PhysicalResource* cannot have repeated elements.
- For a given *TaskSchedule*, the list of *HumanResource* cannot have repeated elements.
- For a given *TaskSchedule*, the number of *PhysicalResources* should be equal to the number of *HumanResources*.

5 Milestone 3 – Production Optimization

This section presents the steps for the achievement of the final milestone. The objective of *Milestone 3* is to create an optimized algorithm, maximizing the resource utilization, and minimizing the total production time.

5.1 Scheduling Algorithm Optimization

In this milestone, the process of import and parse the input files is exactly from the first milestone, same validations, and rules. The input of this algorithm is the same as the *FIFO* algorithm developed on *Milestone 1*.

The main objective of the *Milestone 3* algorithm, as write above, is maximizing the resources utilization, minimizing the total time needed for all task schedules. The result should be a better solution than the output of *Milestone 1*.

There are new rules that need to be applied, a "greedy" optimization is intended, as many tasks as possible are started at any given time. To start the tasks, the set with the smallest task id(s) must be used, for example, if it is possible to start the sets *{TSK_1, TSK_2}* and *{TSK_1, TSK_3}*, the set *{TSK_1, TSK_2}* must be started.

The assigning of *PhysicalResources* to tasks must be assigned sequentially and sorted by ID. The same logic should be applied to *HumanResourses* with the first permutation found that works.

At any given time, tasks from instances of products that are already partially in production are priorities, before trying new instances of products.

Tasks already scheduled must be ordered by *StartTime*, *OrderId*, and *ProductNumber*.

A *Task* of a *Product* instance can only be started if the previous *Task* of that instance has already been finished.

The activity diagram shown in Figure 7 represents an overview of all steps developed for the optimized algorithm:

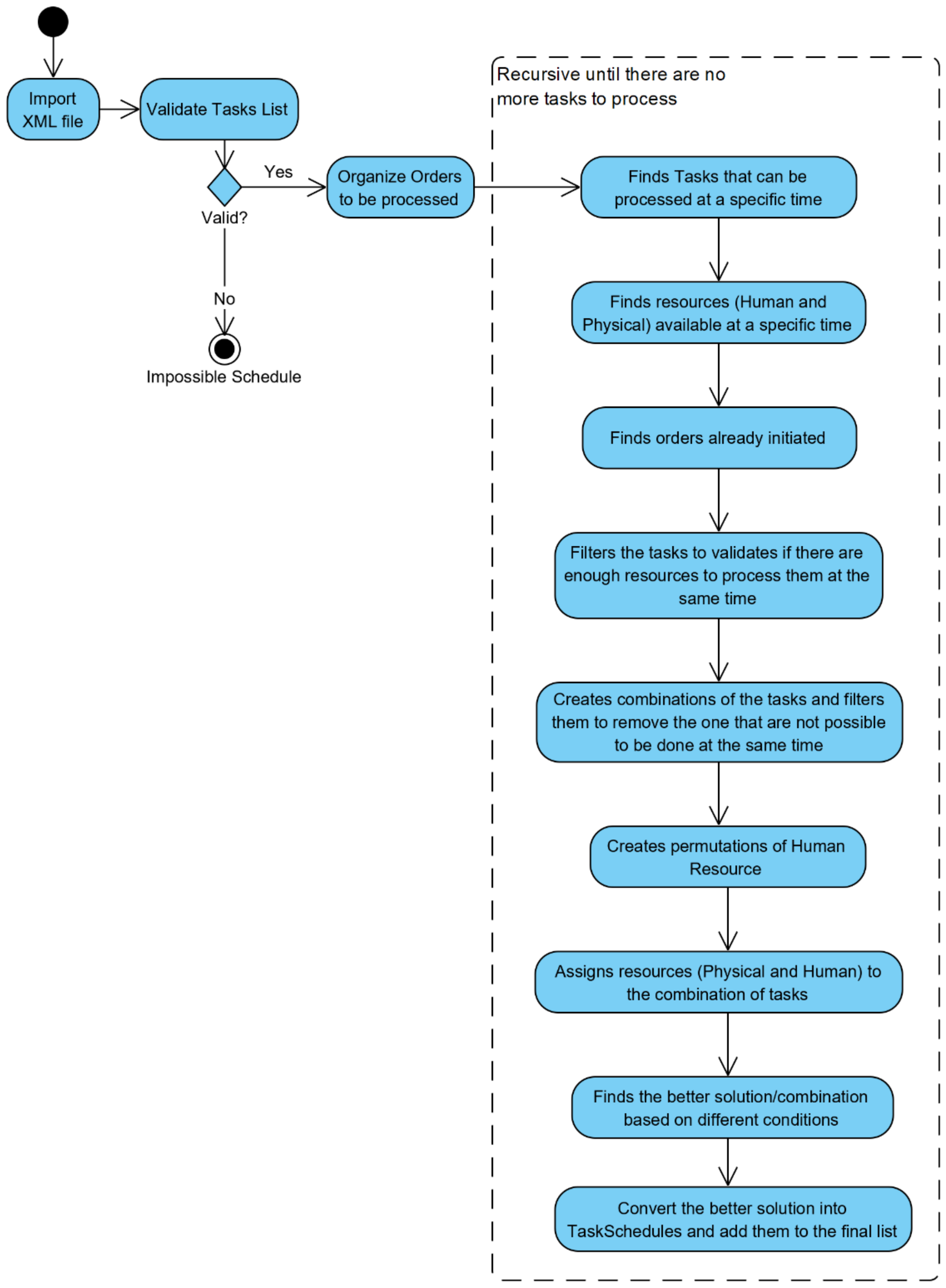

Import XML file
Validate Tasks List
Yes
Valid?
No
Impossible Schedule
Organize Orders to be processed
Recursive until there are no more tasks to process
Finds Tasks that can be processed at a specific time
Finds resources (Human and Physical) available at a specific time
Finds orders already initiated
Filters the tasks to validates if there are enough resources to process them at the same time
Creates combinations of the tasks and filters them to remove the one that are not possible to be done at the same time
Creates permutations of Human Resource
Assigns resources (Physical and Human) to the combination of tasks
Finds the better solution/combination based on different conditions
Convert the better solution into TaskSchedules and add them to the final list

Figure 7 - Flow diagram that represents the milestone 3 algorithm.

It starts by importing the *XML* file and validating whether it is possible to schedule, then the *Tasks* that can be performed at that moment are chosen, and a recursive cycle is entered with a stop condition, the recursiveness ends when all *Tasks* of all *Orders* are in the *Processed* state.
Finally, the *TaskSchedule* is created and added to the final list that will result in the output *XML*.

In order to facilitate the interpretation of the developed code, all the methods have *ScalaDoc* and an *HTML* page was automatically generated with all the information of the developed methods, signature, parameters, and feedback.

5.2 Tests

To ensure the quality and safety of the methods developed for the algorithm of this milestone, tests like those described in section 3.4 were implemented.
The *TaskStateModelListTests* and *MS03SchedulerTests* classes have been added with new unit tests.
It is relevant to compare the results obtained by the *Milestone 1* algorithm with the outputs of this milestone, the execution of the artifact of this milestone should provide a production time equal to or smaller than the artifact of the first milestone. With the purpose of validating this, *SchedulerTests* class was created, comparing both algorithms' output when given the same input.
At least, for function tests, class *ScheduleMS03Tests* was created.

6 Conclusion

This chapter closes the report by presenting an overview of the milestones developed. Possible future improvements are described and a discussion of achieved goals is presented.

6.1 Future Improvements

To improve the readability and facilitate the understanding of the code, should have been created some objects that group several elements instead of using a *tuple*. A *tuple* is an ordered collection of objects of different types [3].
Another noticeable improvement is making all the provided test files pass the test execution. It is understood that some criteria described in section 5.1 are not being correctly considered.
Furthermore, it would be interesting to include load tests in the project to validate the algorithm breaking point in terms of how many orders can it schedule in usable time.

6.2 Discussion

The algorithms implemented in the different milestones fulfill the intended objective nevertheless, there are always opportunities to improve the presented implementation, as described in the previous section.
As expected, milestone 3 presented an added difficulty, considering all the requirements and development times available.
The group considers that there are requirements for the optimized *Milestone 3* algorithm that should be better developed and clearer to interpret. Sometimes solutions were found that apparently would be a better solution than the output provided in the project repository.
The objective across the entire project was successfully achieved, it was proved that the optimized algorithm has better solutions than the *Milestone 1* algorithm.

6.3 Final Conclusion

After implementing all three milestones the team believes that the last milestone had the biggest complexity since the requirements were more complex. And also, the algorithm produced had more iterations and validations.
Overall, all three milestones were accomplished with success. This project achieved good results in every milestone and the members worked well as a team.

7 Bibliography

[1] "GeeksForGeeks," 16 April 2019. [Online]. Available: https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/.

[2] N. Malheiro, "Moodle ISEP," [Online]. Available: https://moodle.isep.ipp.pt/pluginfile.php/96639/mod_resource/content/7/TAP_T_FP_W6_1_PBT.pdf. [Accessed June 2021].

[3] Baeldung, "Baeldung Scala," 7 May 2021. [Online]. Available: https://www.baeldung.com/scala/tuples. [Accessed 19 June 2021].