Production Schedule

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.

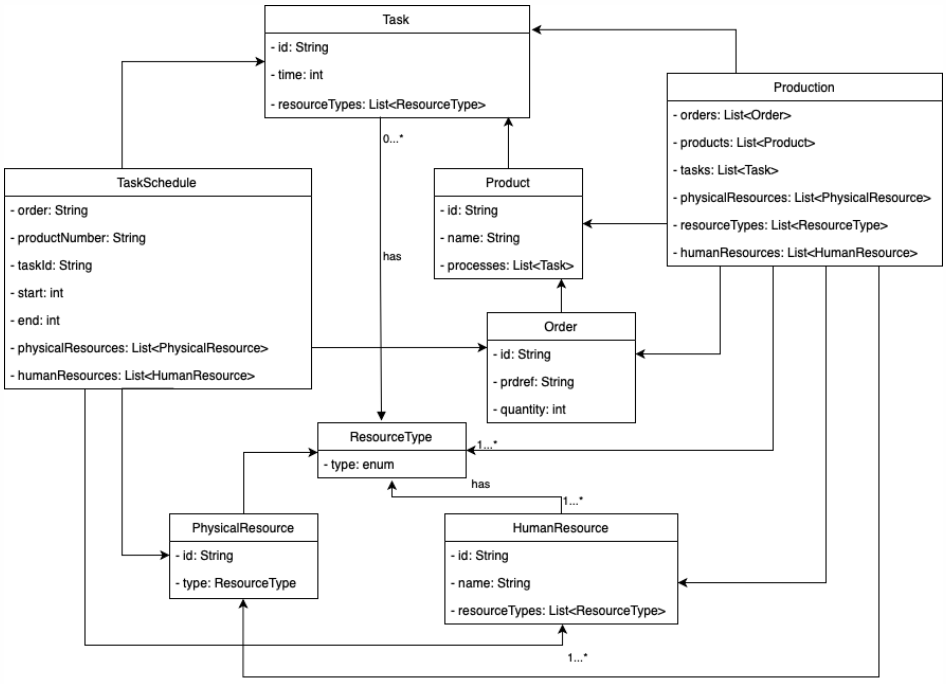
The development was divided in three milestones, being the first one a Minimum Viable Product (MVP) implementation of the scheduling algorithm, followed by the development of property-based testing for the domain, and the third one, a refinement to the scheduling algorithm with scheduling optimization.

Team Members:


```
- Rute Santos (1160663)
- Filipe Ferreira (1160826)
- Vera Dias (1160941)
```


Domain Concepts

After analyzing the requirements described in the statement, the following domain model was built:



Class	Description
HumanResource	Entity that contains all the infomation for a Human Resource, it's 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*.
TaskSchedule	The final result of running this application should be a set of *TaskSchedule*. This entity presents all the *Human* and *Physical*, necessary for a given *Order* of a *Product*. The required time interval is represented by *StartTime* and *EndTime*.
Production	The Production is the entity that contains everything necessary to create a *Schedule*. The Production has a list of *Orders*, a list of possible *Products*, a list of *Tasks*, a list *Physical Resources*, a list of available *Resource Types* and a list of *Human Resources*. To create a Schedule, it's necessary to contact Production to know all the resources available.

Domain Validations

Input validations are required for entity identifiers. The following Regex expressions were used:

Class	Validations
HumanResource	- ID must follow the pattern `HRS_[1-9][0-9]*` - Name must not be null or empty
Order	- ID must follow the pattern `ORD_[1-9][0-9]*` - Quantity must be bigger than 0
PhysicalResource	- ID must follow the pattern `PRS_[1-9][0-9]*`
Product	- ID must follow the pattern `PRD_[1-9][0-9]*` - Name must not be null or empty
ResourceType	- Must follow the pattern `PRST [1-9][0-9]*`
Task	- ID must follow the pattern `TSK_[1-9][0-9]*` - Time must be bigger than 0

Milestones

Milestone 1 (MS01)

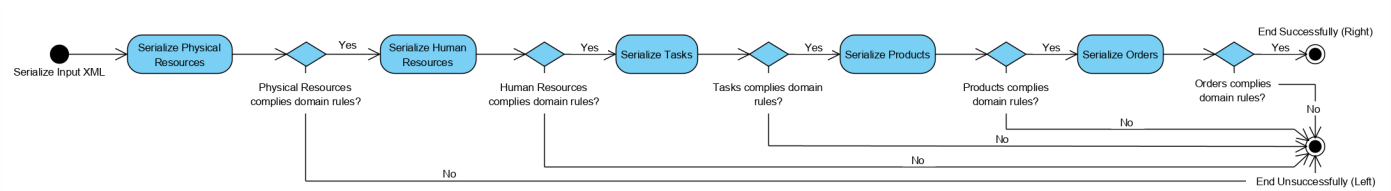
XML File Parse

For this iteration there are two main flows using XML, the (i) input and (ii) output.

Input

This part starts in the *ScheduleResolver* class. For this, the input xml is used to create a *Production* object from a XML Node. To create a *Production* object it's 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 serialize the *Physical Resources*, if this does not comply to the domain rules (described in the *Domain Validations*) then the serialization is unsuccessful and it's generated a error in the file. In the case that the *Physical Resource* is valid then it's necessary to Serialize the Human Resources that will follow a similar flow described above. After this, the *Tasks* serialized (this will fail if the tasks do not comply the domain rules), after this the *Products* are serialized 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 a *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 following diagram represents the flow of serialization before the algorithm begins.



Output

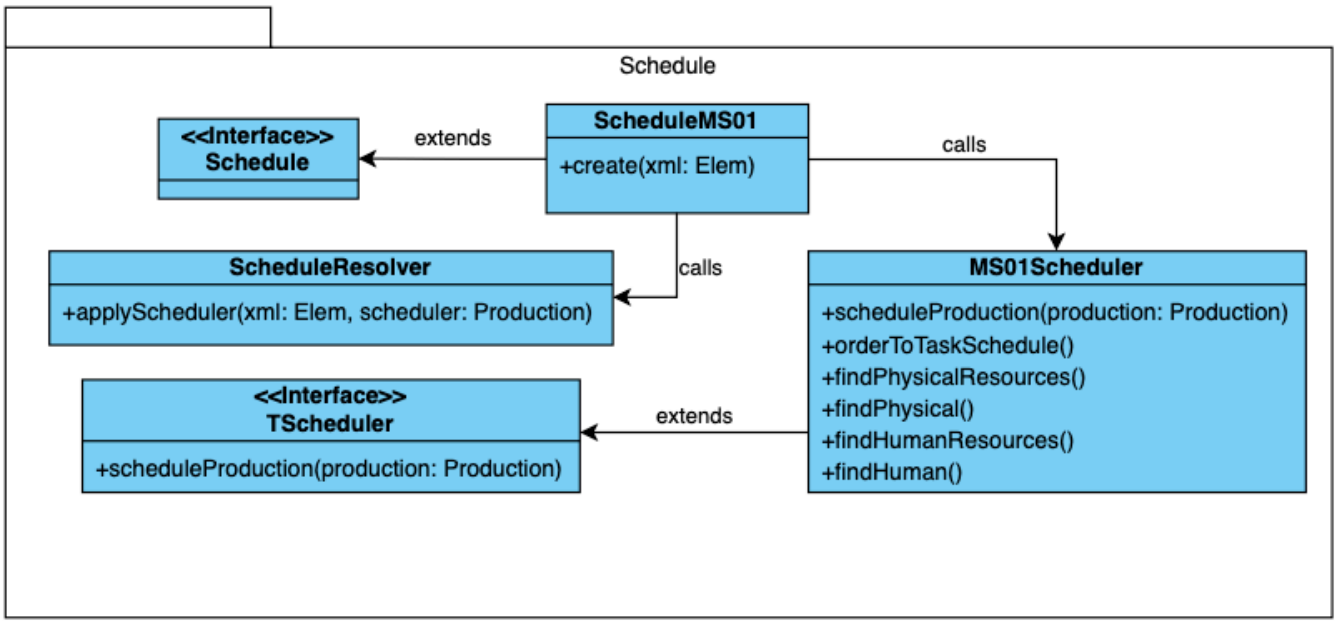
After the algorithm generates a list of *Task Schedule* it is necessary to generate a XML file that has each individual element of Task Schedule. Each element is serialized using a auxilary method in the *XMLParser* class. For each Task Scheduler element there is a *order id*, *product number*, *task 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 a 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 a error occurs along the whole process the serialization does not occur and a DomainError is returned instead of a Elem.

Scheduling Algorithm

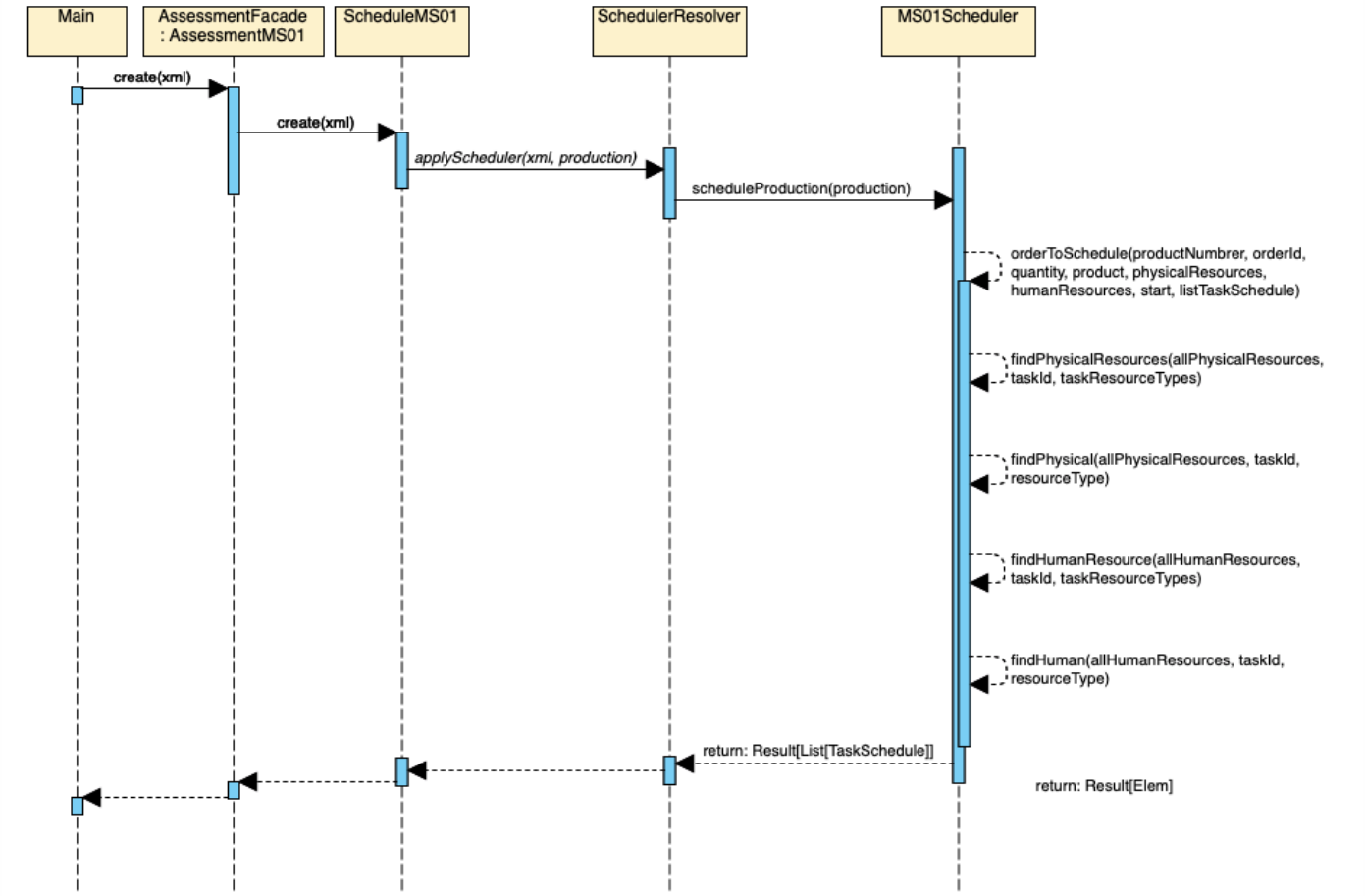
The development of the algorithm is distributed among different classes of the package **Schedule**:



The algorithm for the creation of **Schedule** uses *tail recursion* and consists on 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*.

The diagram below describes the phases of execution of the algorithm:



Future Improvements

There are some aspects that should be improved on the next iterations, such as:

- A service layer should be created to accomodate auxilary methods that are used in the domain layer. Some methods that would fall in this category are *findHuman* and *findPhysical* in MS01Scheduler class;
- It's a good practice to add explicit type notation. This should be used whenever possible;
- There are some warnings that are shown when the program is compiled using the terminal. These warnings should be assessed;
- Finally, also as a good practice, the use of *head* should be avoided.