Milestone 1

Técnicas Avançadas de Programação

1180761 – Jorge Pessoa

1181597 – Rui Alves

1191421 – Luís Neves

Glossary

1. **Agenda** – Information block containing the existing runways, the aircrafts intending to land and the maximum delay time a regular aircraft can experience.

2. **Aircraft** – Vehicle that will need assignment to a designated runway based on a series of parameters. An Aircraft is characterized with an *ID*, an *Aircraft Class*, a *Target Time,* and an optional *Emergency Delay*. The *Target Time* is the base reference for the ideal time of an aircraft operation to occur, this can be delayed by the maximum time delay defined in the *Agenda*. If an aircraft has an *Emergency Delay* value, it means it's in an emergency setting and cannot be delayed more than the specified value.

3. **Runway** – Airport runway characterized by an *ID* and a list of *Aircraft Classes* it can handle.

4. **Aircraft Class** – Characterizes an aircraft based on its type of operation (*Landing* or *Taking-Off*) and its weight (*Small, Large* and *Heavy*). This is defined at the start of the project requirements, as we can see in Figure 1.

Number	Operation	Weight
1	Landing	Small
2	Landing	Large
3	Landing	Heavy
4	Take-Off	Small
5	Take-Off	Large
6	Take-Off	Heavy

Figure 1 - Aircraft Class Definition

5. **Separation of Operations** – Set of rules defined at the start of the project requirements that define the mandatory necessary time between operations in a runway. As seen in Figure 2, the different types of operations between different types of aircrafts makes a big difference in the runway assignment algorithm.

			Trailing aircraft					
			Landing			Take-off		
			Small	Large	Heavy	Small	Large	Heavy
Leading aircraft	Landing	Small	82	69	60	75	75	75
		Large	131	69	60	75	75	75
		Heavy	196	157	96	75	75	75
	Take-Off	Small	60	60	60	60	60	60
		Large	60	60	60	60	60	60
		Heavy	60	60	60	120	120	90

Figure 2 - Separation of Operations Table

6. **Schedule XML** – One is created for each assigned *Aircraft*. It is characterized by the Actual Target the *Aircraft* got to land, the associated *Cost* in case there was a delay, a *Runway* and of course an *Aircraft*.

Domain Model

As seen in Figure 3, the domain model for the present project is relatively simple, where all the above mentioned connections and relations are present.

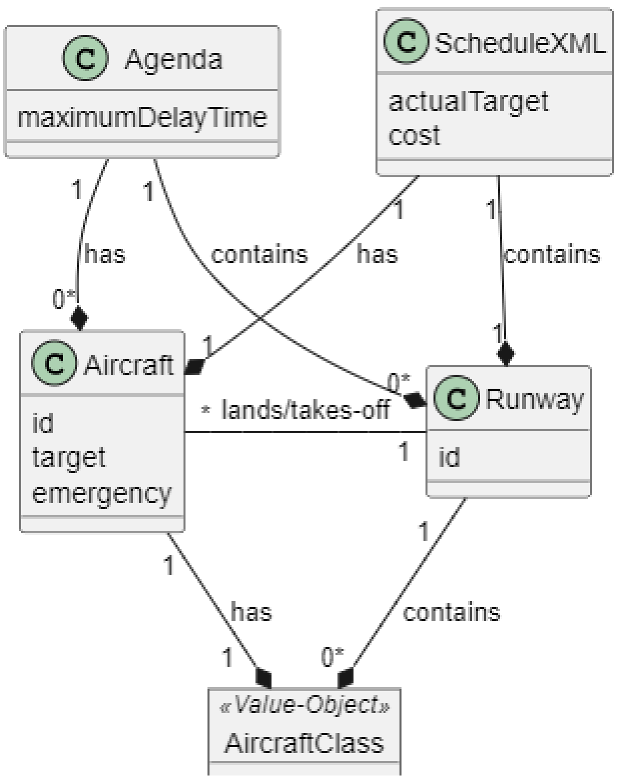

Figure 3 - Domain Model

As seen above, we can extract the main objective of the assignment, which is to elaborate a *Schedule* for each aircraft present in the *Agenda* while taking into account all the necessary restrictions, such as the maximum delay, the specific *Aircraft Classes* a *Runway* accepts and so on.

Domain Implementation

When thinking of and designing the development process for the project, the team thought of the best practices one can take to assure domain level correctness and validation.

To do so, the team implemented opaque types for domain rules validations, such as the implementation of a *Non Negative Integer*, a *Positive Integer,* or an *Identifier*, as seen in Figure 4, where there's an example of the *Non Negative Integer* type.

```
opaque type NonNegativeInteger = Int

Jorge Pessoa +1
object NonNegativeInteger {
  Jorge Pessoa
  def apply(number: Int): Result[NonNegativeInteger] =
    if (number >= 0) Right(number)
    else Left(IllegalNonNegativeInteger("Invalid NonNegativeInteger"))

  Luís Neves
  def lessThan(number: NonNegativeInteger, numberCompare: Int): Boolean = number < numberCompare

  Jorge Pessoa
  extension (m: NonNegativeInteger)
    Jorge Pessoa
    def -(n: NonNegativeInteger): Int = m - n
    Jorge Pessoa
    def +(n: Int): NonNegativeInteger = m + n
    Jorge Pessoa
    def NNItoInt: Int = m
```

Figure 4 - Opaque Type Implementation

This concept is visible and implemented throughout the algorithms and adjacent methods, as seen in Figure 5.

```scala
final case class Aircraft(id: Identifier, aircraftClassParam: AircraftClass, target: NonNegativeInteger,
                          emergency: Option[PositiveInteger])

Jorge Pessoa
object Aircraft:

  val id: String = "id";
  val class_number: String = "class";
  val target_time: String = "target";
  val emergency: String = "@emergency"
Luís Neves +1
def parseAircraft(xml: Node): Result[Aircraft] =

  for {
    id <- fromAttribute(xml, id).flatMap(Identifier.from)
    cls <- fromAttribute(xml, class_number).flatMap(aircraftClass)
    target <- fromAttribute(xml, target_time).flatMap(a => NonNegativeInteger(a.toInt))
    emergency <- (xml \ emergency).headOption match
      case Some(emergencyValue) => PositiveInteger(emergencyValue.text.toInt).map(Some.apply)
      case None => Right(None)
  } yield Aircraft(id, cls, target, emergency)
```

Figure 5 - Aircraft Implementation

To simplify the process of consulting the hardcoded values for the Separation Rules and the Aircraft Classes the team implemented a Utils class to return the necessary value for the required Aircrafts, as seen in *Figure 6*

```scala
def aircraftSeparation(leading: AircraftClass, trailing: AircraftClass): Int =
  (leading, trailing) match
    case (Landing(Small), Landing(Small)) => 82
    case (Landing(Small), Landing(Large)) => 69
    case (Landing(Small), Landing(Heavy)) => 60
    case (Landing(Small), _) => 75

    case (Landing(Large), Landing(Small)) => 131
    case (Landing(Large), Landing(Large)) => 69
    case (Landing(Large), Landing(Heavy)) => 60
    case (Landing(Large), _) => 75

    case (Landing(Heavy), Landing(Small)) => 196
    case (Landing(Heavy), Landing(Large)) => 157
    case (Landing(Heavy), Landing(Heavy)) => 96
    case (Landing(Heavy), _) => 75

    case (TakeOff(Heavy), TakeOff(Large)) => 120
    case (TakeOff(Heavy), TakeOff(Small)) => 120
    case (TakeOff(Heavy), TakeOff(Heavy)) => 90
    case (_, _) => 60
Fausto21 +2
def aircraftClass(classNum: String): Result[AircraftClass]  = classNum match
  case "1" => Right(Landing(Small))
  case "2" => Right(Landing(Large))
  case "3" => Right(Landing(Heavy))
  case "4" => Right(TakeOff(Small))
  case "5" => Right(TakeOff(Large))
  case "6" => Right(TakeOff(Heavy))
  case _ => Left(IllegalClassNumber("Identifier cannot be empty"))
```

Figure 6 - Utils Class

Algorithm Implementation

In this section, we will describe the logical process associated with the elaborated algorithm in smaller, more comprehensive steps.

1. Load information from XML file. Using the parsing methods implemented in the respective classes, the process starts by loading the information to create an *Agenda*, as seen in Figure 7.

```scala
def create(xml: Elem): Result[Elem] =
  Agenda.fromXML(xml) match
    case Left(a) => Left(a)
    case Right(b) => generateSchedule(b) match
      case Left(a) => Left(a)
      case Right(b) => Right(exportToXML(b))


Jorge Pessoa
def generateSchedule(agenda: Agenda): Result[List[ScheduleXML]] =
  for {
    _ <- validateAgenda(agenda)
    res <- assignRunway(agenda)
  } yield res
```

Figure 7 - Initial Methods

1.1. The algorithm then starts by validating the received *Agenda* object and its children to catch any errors in the XML formatting, such as repeated *IDs*, or unassignable *Aircrafts*. As seen in Figure 8.

```scala
private def validateAgenda(agenda: Agenda): Result[Unit] =
  for {
    _ <- validateAircrafts(agenda.aircrafts)
    _ <- validateRunways(agenda.runways)
    _ <- validateAvailableRunways(agenda)
  } yield ()

Jorge Pessoa
private def validateAircrafts(la: List[Aircraft]): Result[Unit] =
  la.map(_.id) match
    case ids if ids.distinct.sizeIs == ids.size => Right(())
    case ids => Left(DomainError.RepeatedAircraftId(ids.diff(ids.distinct).map(Identifier.identifierToString).mkString(", ")))

Jorge Pessoa
private def validateRunways(la: List[Runway]): Result[Unit] =
  la.map(_.id) match
    case ids if ids.distinct.sizeIs == la.size => Right(())
    case ids => Left(DomainError.RepeatedRunwayId(ids.diff(ids.distinct).map(Identifier.identifierToString).mkString(", ")))


Jorge Pessoa +1
private def validateAvailableRunways(a: Agenda): Result[Unit] =
  val supportedClasses = a.runways.flatMap(_.classes).distinct
  val unsupportedClasses = a.aircrafts.map(_.aircraftClassParam).filterNot(supportedClasses.contains)

  unsupportedClasses match
    case Nil => Right(())
    case unsupported =>
```

Figure 8 - Domain Validation

1.2. We then call the *assignRunway* method that takes an *Agenda* object as parameter and defines a tail recursive method that takes a list of Remaining Aircrafts and a list of Assigned Aircrafts. The list of Assigned Aircrafts is initialized as empty.

1.2.1. We start by iterating over the list of *Aircrafts*

1.2.1.1. Use a helper method to get runways that can accommodate the current *Aircraft.* Returns error if no runways can accommodate the *Aircraft*.

1.2.1.2. Iterates over available *Runways*.

1.2.1.2.1. If no Aircrafts have been assigned to the current Runway, assigns aircraft to current runway and calls loop again.

1.2.1.2.2. If Aircrafts have already been scheduled, check the availability of the rest of the Runways. If any Runway is empty and available to receive that Aircraft, assign to that one.

1.2.1.2.3. If there are scheduled Aircrafts and there are not empty Runways, the algorithm then tries to assign the Aircraft to the Runway that will cause it the least delay using the *findBestRunway* method.

1.2.1.2.3.1. The method starts by elaborating a List of Tuples containing the Runway and the required time to assign the Aircraft to it, after that, it iterates through the list and finds the smallest time value, thus finding the best Runway.

1.2.1.2.4. With the Tuple return of the previous method the algorithm now calculates the delay that will be applied to the aircraft.

1.2.1.2.4.1. If the delay is less than the *Agenda*'s Maximum Delay Time, the algorithm calculates the cost of the delay and the Aircraft's updated target time.

1.2.1.2.5. The algorithm then checks to see if this is an Emergency Aircraft and if the calculated delay is bigger than the allowed Emergency Delay, the algorithm calls the method *findFitForEmergency*, since the Emergency Aircraft has priority over the rest of the regular Aircrafts.

1.2.1.2.5.1. This method takes as parameters the list of assigned Aircrafts and the Emergency Aircraft to be assigned and defines a tail recursive method inside that takes as parameter the remaining list of Assigned Aircrafts, the list of Aircrafts that were assigned but will be removed and the list of Aircrafts that are deemed untouchable.
For example, if we want to add an emergency Aircraft to this list that represents a Runway, the red colored n's will be

untouchable since they would alter the emergency Aircraft that is already in that Runway because they were scheduled before it.

[normal, n, n, Emergency 1, n, n, n] <------ Emergency 2

On the other hand, the green n's are alterable since they were scheduled before.

1.2.1.2.5.2. The method then evaluates if the Runway Interdict List does not contain the current Aircraft's ID. Because if it does, we need to skip to another runway, since that one is already searched and not available for our needs.

1.2.1.2.5.3. Then, the necessary separation time is calculated between the Aircraft in question and the Emergency Aircraft.

1.2.1.2.5.4. We then calculate the time interval the Emergency Aircraft can do. That is, the Emergency Aircraft Target + the Emergency Target. For example, for a Target of 50 with an Emergency Target of 5, the interval would be [50, 55].

What the method does is calculate if the current Aircraft + it's Separation Rule is less or within those boundaries. If it's less than the lower boundary there is no delay to the Emergency Aircraft, if its between the boundaries, it calculates it's delay and assigns the Emergency Aircraft to the Runway with the updated Target.

If it's above the higher boundary, the current Aircraft needs to be unassigned (removed from the list of Assigned Aircrafts) and the loop is called again, now with the next Assigned Aircraft to make the same calculations and see if the Emergency Aircraft can be assigned. Worst case scenario, the Emergency Aircraft substitutes every Aircraft and takes first place in the Runway.

When it finds a spot for the Emergency Aircraft, it returns and exits the loop.

1.2.1.2.5.5. After assigning the Emergency Aircraft to a Runway, the regular calculations need to be made, such as the cost.

1.2.1.2.5.6. After that, the algorithm verifies if the removed Aircraft was the first in the Runway, to then assign the Emergency Aircraft as the first.

1.3. When the loop exits, we then catch the correct output and write the list of Schedules to a file.

Testing

The group was able to complete 18/20 tests that came with the project, making a final score of 90.
Regarding other tests, the group implemented Unit and Functional Tests.
As seen bellow, in Figure 9, there’s an example of a Unit Test for the creation of an Aircraft Class.

```
class AircraftTest extends AnyFlatSpec with Matchers {

  "parseAircraftEmergency" should "parse valid XML into an Aircraft object" in {
    val xml =
        <aircraft id="AC1" class="1" target="139" emergency="5"/>

    Identifier.from( id = "AC1") match
      case Right(value) => aircraftClass( classNum = "1") match
        case Right(airClass) => NonNegativeInteger(139) match
          case Right(tg) => PositiveInteger(5) match
            case Right(em) => Aircraft.parseAircraft(xml) shouldBe Right(Aircraft(value, airClass, tg, Some(em)))
            case Left(_) => DomainError.IllegalNonNegativeInteger("")
          case Left(_) => DomainError.IllegalPositiveInteger("")
        case Left(_) => DomainError.IllegalClassNumber("")
      case Left(_) => DomainError.IllegalIdentifier("")
  }
```

Figure 9 - Unit Tests