Instituto Superior Técnico

Object Oriented Programming

Prof. Alexandra Sofia Martins de Carvalho

Final Report

Project 17/18

Group 2

Name:
António Silva
João Guerreiro

Number:
81628
81248

May 9, 2018

1. Goals

This project focuses on the implementation of a specific Stochastic Optimization Problem using Java.

Being Java an OO Language we can immediately see that even tho our project topic is specific, our solution should be extensible and reusable to other Optimization Problems or even other applications.

With this said we will try to use most of the knowledge acquired during the lectures of the course, all the way from Inheritance and Polymorfism to Interfaces and even the java.utils library. To test and deepen our knowledge we will also try to use generic types, for example, even tho we could easily do the project without those.

In the present report we will start out by talking a bit about our solution and implementation, give a small overview of the UML, give a critical evaluation of the developed code and its performance and finally some conclusions.

2. Implementation

2.1 Main Structure

Our project is divided in 4 packages.

- main – used to test our specific problem.
- optProblem – the actual implementation of our optimisation problem.
- parser – parses the provided files needed for our project.
- pec – generic pending event container.

There are several aspects that lead to our differentiation of the code in different package. We started out with the optProblem package that is the solution for the problem in the project paper. We then decided to separate between the PEC and this last one since the Pending Event Container can be used in applications outside the Optimization Problem scope and thus should be in a package of their own. The parser was placed in a new package in the end since its functionality is to parse a XML file that goes alone the simulation.DTD but this can be done not only in an Optimization Problem. Lastly we got the main package that is only used to test our program and nothing else and because of that it makes sense to place it on it's own.

2.2 Major Classes in the optProblem

Our project has 4 big classes in the optProblem package:

- Map – used to store the map information.
- Individual – the actual individual that will move around the grid.
- Event – Things that can happen during the simulation.
- StochasticOptProblem – Solution to the problem in the project paper.

These 4 are the main classes of our program but we can't forget about others like the Point or SpecialZone. Those won't be referred in the present report since they are really straight forward and there is no need to explain the choices made. They can easily be seen in the UML as well.

Starting with the Map Class, this one was nonexistent until a late phase of the project. Its attributes used to be part of the Optimization Problem. We then realised that a Map is a more general concept and the same Map can be used in different Optimization Problems. We should also make reference to the fact that the map class implements the map Interface to make it even more reusable. It's important to say that some attributes like the starting point or end point are not part of the map. This was our choice since it means we can run the same map (essentially the grid) with different positions. These are in the Optimization Problem as "Map related fields".

The Individual class is used for the Individual type objects. These should have an

unique identifier that allows us to keep track of who is who. They also need a Death Date to be assigned. This makes it easier when we need to create a next move or a next reproduction since we can check directly if the time of the event is before death ro after death. As we did for the Map, some attributes that we can at first relate to individuals like the starting number of individuals or maximum number of individuals were by us chosen to be part of the optimization problem instead since they are specific of a certain optimization problem. The way of putting these in the individuals and having them not carry each this attribute would be putting it as static but this would prevent from several optimization problems to be run at the same time. These are in the Optimization Problem as "Individual related fields".

The Event Class is abstract and is extended by the events of the types: Move, Reproduction, Death, Epidemic and ControlPrint. Each of this events override the methods in the abstract class and give their own implementation of the ExecEvent method. What each event type does is pretty self explanatory so we'll leave it out of the report. However we should say once again that the means for the event exponential are not a static field of the Event class following the same line of thought of the 2 previous classes. This means that if we had done it a static field of Event we wouldnt be able to have different optimization Problems with different mean parameters running at the same time since the attribute would be a characteristic of the class itself and not the objects. These are in the Optimization Problem as "Event related fields".

The StochasticOptProblem is now easy to comment since basically it gathers everything needed to run a specific problem. Several optimization problems can be ran at the same time if thats what we intended since the static fields in the classes are not specific to each optimization problem.

The other particularities of these classes and the others not described here can be seen in the source code comments as well as the Javadoc.

2.3 Reusability

We ensure the reusability of our project through various means:

- Separating the PEC and optProblem packages, we allow the PEC package to be used in other problems.
- PEC is implemented as a generic type meaning we can pass any Type extending Object. The PriorityQueuePec has the <T extends IEvent> meaning that we are restricting the object we can receive to subclasses/implementations of the interface IEvent.

- Polymorphism: An event can take many forms. We use 5 possible ones and all one has to do is implement IEvent.
- Various interfaces were implemented to allow, for example, extending the optProblem with new events or different map types.
- Encapsulation: Most attributes - if not all in some cases - are set to private and can be accessed or modified via getters and setters - this can be seen in the UML or in the section with the same name of the present report.

An example of the reusability of our code was seen when implementing our solution. We started out by having only 3 events: Move, Death and Reproduction. Then we decided we should add Epidemics as events with instant time as well as the Control Prints as events with $T_{total}/20$ time increments. All we had to do was extend the abstract Event class that implements the interface IEvent and add the events to the PEC. Everything else worked smoothly since we only need events to follow that structure.

This is especially evident if we consider the other Optimization Problem example given by the teacher. In that one we also had events - that could very well be implemented with this interface and this Pec - even tho the concrete events are very different, for instance ours have an individual associated.

In the same line of thought we can look at the interface for the OptimizationProblem. By defining this interface as general as it can be we are allowing other users to do their own implementation of an Optimization problem by defining the 3 bigger methods: Initialize, Simulate and Run.

Having the attributes on private was something we changed late in the game but still proved usefull. One of the times we got to se this was when we changed how we were storing the Special Zones (from list to array). Since we were using getters/setters to access these in the map the transition was smooth since we only had to change those methods instead of re-writting all the code that was accessing the variables. This also means that one can later make a subclass and overwrite the getters/setters methods if needed to achieve different results and all the code will work perfectly.

2.4 Wrong XML Entries

There are several places where we check for the arguments passed in the XML file.

- If the starting Point is outside the grid then it is assumed individuals start at (1,1).
- If the final Point is outside the grid then it is assumed individuals should reach the end of the grid (N,M).

- If the number of Obstacles is X and we try to pass more than X Obstacles than the extra entries will be ignored.
- If the number of Special Zones is Y and we try to pass more than Y Special Zones than the extra entries will be ignored.

After having the parser completed we encountered a problem: What if the user messes up the XML file but in a way we can't detect with the DTD? Our solution was the one proposed in the items above. Basically the idea is having a set of default values that are used if the person doesn't follow the correct format.

2.5 Automated Testing

We tried to use JUnit 5 to test our program automatically. These tests are present in the *main* package and invoke 4 of our test files. The expected output of the optimisation problem is compared to the actual output which we then use to determine the continued reliability of the program.

It is important to give special attention to some results since this is a Stochastic Simulation. With that said one cannot assume that the implementation is not correct simply because the given output of one simulation is not that same as the optimal path. Only by running the tests multiple times - in the bigger tests it can be quite time consuming to run them a lot - we can have an idea about the actual best path and if the program is working properly or not. With this said, we included the JUnit 5 part so we could deepen our knowledge when it comes to Java practices.

3. UML - Class Diagram

Even tho the full resolution UML is provided in the folder specified in the project paper, we thought we should give a brief introduction to what was being shown.

The first thing we notice is the 4 different packages already mentioned in the current report. We gave special attention to the relationship between the different classes and interfaces. This means we tried to replicate in our code what was described in this UML Diagram. Some things were later changed in the code since we only then realised they were better implementations and so we changed the UML accordingly.

Since most of our attributes are private - something also already discussed - there was a huge need to use getters and setters. The attributes that use these have the «Property» tag before them. This is done in Visual Paradigm so the UML doesn't get insanely large due to the big amount of get/set methods. This way, if a method has «Property» before its visibility and name it means it has at least a getter or a setter that was omitted in the methods part of the Class Diagram.

Some of the names names of the associations might be above the corresponding arrows/lines because every small thing we changed out dis-format a lot of what was already done. We also tried to show in different associations the cases where class A has 2 attributes of class B like b1 and b2. We hope we covered all of these and none are associations with multiplicity 2 like we started with.

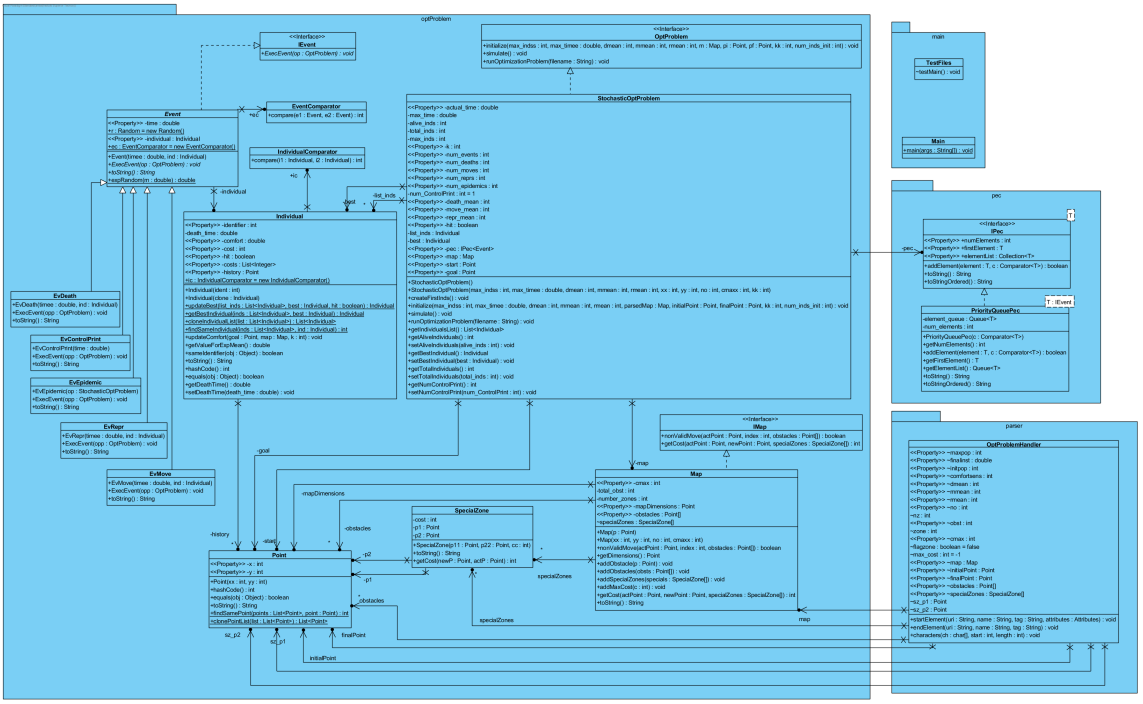

Figure 3.1: UML of the Solution Implemented

4. Critical Evaluation

In this chapter we will try to resume what was said in the whole report by looking as outsiders to our solution and its performance.

4.1 Developed Code

As we said several times through the report, the solution is extensible/reusable. The most reusable parts are the PEC and Events. This is also the place where the polymorfism is more evident. The code also allows for several simulations to be run in parallel as the data structures only hold information needed for the said simulation: Individuals, Events or the Map do not contain attributes relative to the Optimization Problem they belong to.

The code is commented where we thought extra information should be given to ease the understanding of the solution. As requested in the project paper we also provide the javadoc type comments to generate the Javadoc included in the zip folder.

Some Java(ish) features learned in the lectures that we tried to include were:

- Different packages;
- Attributes visibility;
- Comparators for the event and individual;
- Several interfaces;
- Abstract classes;
- Associations;
- Iterating over arrays/lists using the *for* optimized for the iterators;
- Inheritance;
- Generic Types;
- Sax Parser;
- etc.

As the teacher said several times, we also gave special importance to the data structure used for the PEC. We went with the Priority Queue since its implementation is fairly straight forward and it orders the elements it contains using the comparator passed as argument.

Also very important is the fact that we tried our best to have most attributes as private and use getters and setters. This is important not only for security reasons (we only allow someone using our code to access the attributes in a way we defined) but also because any addition we want to do to our code when it comes to getting or setting that attribute (lets assume for example verifying if the arguments passed are correct or performing some sort of mathematical operation) we just have to change the methods and not every instance of object.attribute in the code.

Based on all this we think our implementation checks the criteria of a java solution and it is easy to read and understand.

4.2 Performance

We are going to start this section by talking about the test files.

As we were asked to do in the project paper, we created 5 test files that go with the format specified by the simulation DTD. In these 5 tests we tried our best o make use of different values for the various fields, always respecting the DTD.
The test 0 is the same as the project paper.
The tests 1-3 are simple tests that have different values for the various fields in order to check the correct behaviour of the program developed. The 4th is a bit trickier. In this one we decided to push the grid size as well as the total simulation time (to allow some individuals to reach the goal) and the max number of individuals as well. The 5th test is equal to the first one but with extra entries on the obstacles and zones and the starting and end points are ouside of the map boundaries. This way we test the changes made to the parser for these cases that can't be detected simply by comparing with the DTD format and that shouldn't crash the program since there are easy ways to get around the problem.

Even tho we only send 5 tests in the zip folder, we changed these multiple times and used probably over 20 tests to test our program. From these we identified 4 major factors that limit the application performance:

- Maximum number of alive individuals at a given instant;
- Relationship between the Death and Reproduction means;
- Relationship between the Death and the Move means;
- Total simulation running time.

The first 3 are specially *bad* for our program since they increase by a lot the number of events in the PEC and consequently number of events that need to be dealt with during the simulation. The maximum number of individuals is pretty obvious since the more individuals we have the more events there will be and the PEC will get bigger and bigger. The Relationship between the Death mean and the other 2 is also important since if the Death is really far away an individual will be alive for a longer time and 1) perform more moves 2) have more children. This will also result in the PEC having a lot more events.

The last one could be a determining factor for the memory used by our program if we ignored this issue and will be discussed in the following subsection.

We can now talk about run time performance and used memory performance. They both depend on the above variables more than any others as we already described and justified.

It is important to notice that the tests were done with only one simulation problem

running and conclusions can (and will) change if we are running more than one.

4.2.1 Run Time

The program runs pretty fast if the overall number of alive individuals is not very large (circa 500 is still good). From here we can see an increase in the running time that is worse than linear but nothing extraordinary. Obviously the solution could never be instantaneous and it makes sense that the more individuals we have, the slower it will run.

4.2.2 Used Memory

We paid special attention to this when writting our code and selecting the data structures used.
In the big test files with a lot of individuals and most importantly a large number for the maximum simulation time, the total number of events and total number of individuals gets out of end. Being this way it is important to dispose of those instead of keeping them in memory. As we know, in Java there is the Garbage Collector so all we have to do is stop pointing to a variable and it will be erased sooner or later. We only keep the alive individuals and all the ones that have passed are lost (and collected by the garbage collector). In the same line of thought we remove completly the done events from the PEC instead of saving them somewhere.

5. Conclusion

In conclusion, we believe our project implements all of the laid out requirements without compromise.

We used the java features useful and relevant to our project and according to our tests, the packages provided are robust and provide enough scope to be extended or replaced.

The application behaves nicely under normal conditions being fast and not taking a lot of memory.

Finally, and again using the test files provided, we conclude that the outputs from our implemented solution are correct.