A Farm Simulation using Swing and Concurrency

Filipe Pires [85122], João Alegria [85048]

Software Architecture
Department of Electronics, Telecommunications and Informatics
University of Aveiro

March 16, 2020

1 Introduction

This report aims to describe the work developed for the first assignment of the course of 'Software Architecture', explaining the overall architecture and describing its components and respective communication channels and elaborating on the adopted solutions for concurrency. We also mention how the work was distributed amongst the authors.

The Java application has the purpose of conducting harvest simulations on an agricultural farm. Along with the technical aspects of the implementation, we also elaborate on the adopted solutions for concurrency. Efforts on making the UI highly usable and the code readable and well documented are also stated here. All code developed is publicly accessible in our GitHub repository: `https://github.com/FilipePires98/AS/`.

2 The Agricultural Farm

Agricultural farms have well defined seasons where different activities must be executed to maintain the business productive. Tasks must be distributed amongst workers and quantities must be calculated and tracked for the correct functioning of the entire farm. As the whole system grows, its complexity and difficulty in management grows as well, so management tools emerge as valuable assets for farmers.

Amongst the many features of such tools, one offers a particularly interesting view of the farm as it simulates its behavior. Such simulations allow farm owners to plan harvests and test strategies to understand which offer the greatest productivity and profit. These simulator tools may be as complex as the farm itself, but allow manipulation of time and other resources without any cost. So, it is easy to understand that solutions of this nature offer value to the farming audience in general.

With this in mind, it was proposed to us to develop an agricultural farm harvest simulator with very simple features in order to apply the knowledge gained during the course. This simulator isn't meant to serve as a final product for an agricultural business, rather it should show the potential of such solutions. As it implements concurrency by design, it ensures scalability for a potential product and offers realistic aspects on the virtual farmers' behavior.

But how exactly is the system organized? There are two main entities: the Control Center (CC) and the Farm Infrastructure (FI). The CC is responsible for supervising the harvest, while the FI is the infrastructure for the agricultural harvest.

2.1 Control Center

The Control Center is where the number of farmers to be used is defined, along with their maximum speed, the amount of existing corn cobs and a special parameter called timeout. Timeout, defined in milliseconds, is a parameter used for regulation of the simulation's execution time and sets the maximum amount of time for each farmer to make a movement.

It is in the CC that users send orders for the virtual farmers to execute. The commands available are:

- Prepare - the selected farmers move to a Standing Area, ready for orders.
- Start - the actual simulation begins and farmers start moving.
- Collect - farmers collect corn cobs from the Granary (where the cobs initially are).
- Return - farmers return to the Storehouse with the collected corn cobs.
- Stop - farmers stop whatever they are doing and return to the Storehouse.
- Exit - simulation ends and the program closes.

2.2 Farm Infrastructure

The Farm Infrastructure is the part of the system that has the virtual components of the farm. It holds four sections of the farm:

- Storehouse - where farmers rest and corn cobs are stored.
- Standing Area - where selected farmers wait for further orders.
- Path - a representation of the field that farmers must cross.
- Granary - where corn cobs are temporarily stored.

FI also supports virtual farmers, which during their lives transit between the following states:

- Initial - resting (blocked) in the Storehouse.
- Prepare - ready for orders (blocked) in the Standing Area.
- Walk - moving in the Path (one by one, in the same order they entered it) towards the Granary.
- Wait to Collect - waiting for orders (blocked) to collect corn cobs in the Granary.
- Collect - collecting corn cobs from the Granary.
- Return - moving in the Path (one by one, in the same order they entered it) towards the Storehouse, with the collected cobs.
- Store - storing the collected corn cobs in the Storehouse.
- Exit - farmer kills itself.

So, as you can see, there is a direct mapping between the commands made available to users and the farmer states.

When deployed, the system offers a Graphical User Interface (GUI) for each entity. CC's UI allows user interaction and FI's UI allows visualization of the simulation in real time. We added to CC's interface a mirror of FI's to allow the possibility of both entities running in different environments far away from each other, while ensuring that the user on CC's side has knowledge of what is happening during the simulation. This and other interface-related aspects will be mentioned in greater detail further ahead.

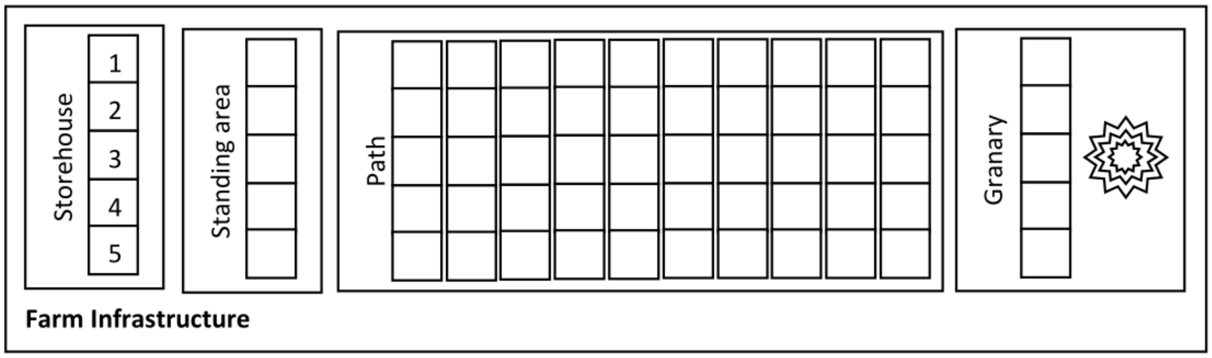


Figure 1: Visual representation of the farm, taken from (*1*).

3 System Architecture

In this chapter we focus on the implementation of each component and on the architecture of the entire solution. Here, we resort to a class diagram (see Figure 2) in UML (*2*) to visually aid interpretation. We also present screenshots of the GUIs to explain how the interaction works.

Our harvest simulator follows a modular structure implemented in Java. As we will see, each component has a specific purpose and is responsible for one key aspect of the system's functionality. The principle of least knowledge is here applied and code dependency is kept low. This means that there are no unnecessary elements but it is needless to say that without any of the components the system will not work as intended.

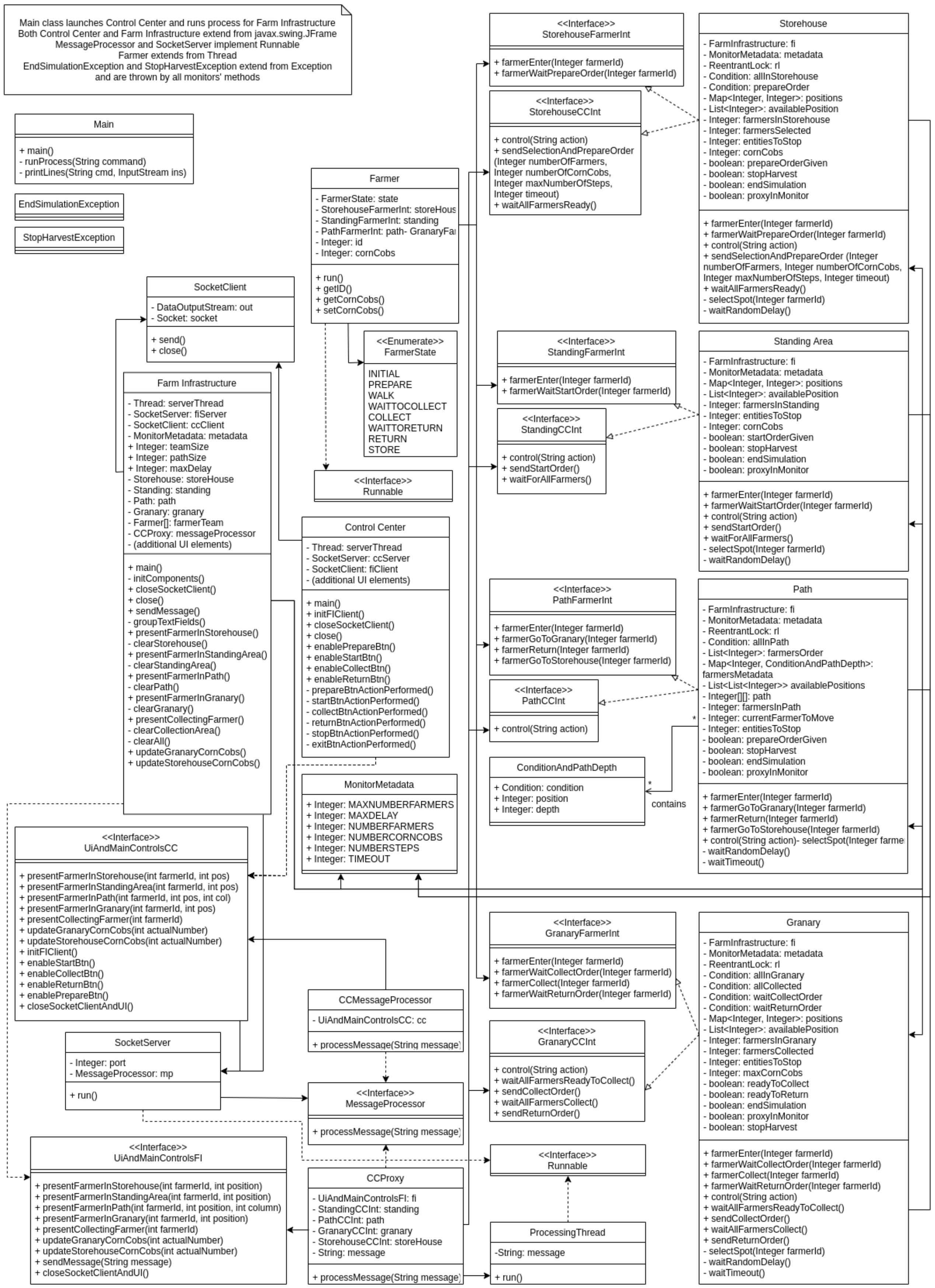
Components belonging to the same logical layer are grouped within a package. Communications between independent components are done through defined protocols. The system is designed to be extensible as well, through the use of well defined interfaces that allow the creation of new versions of farm areas without refactoring the main entities.

All relevant actions during simulations are printed to a unique terminal, with the due source identification, allowing an easy management of what is occurring during execution. To deal with exceptions thrown by user commands, we designed our own exception handling mechanisms. Also, since the user side is the greatest potential source of errors, the interface is confined to its most limited usage, i.e. users are only allowed to do what they can actually do at all times. Although this may seem to reduce the usability of the solution, it actually helps users by guiding them towards what they want to do, without leaving margin for errors.

3.1 Components

Once the Java application is executed, out Main class launches a subprocess dedicated to the FarmInfrastructure and instantiates the ControlCenter, passing full control to it. Every Java application has a single instance of class java.lang.Runtime that allows the application to interface with the environment in which it is running. By obtaining the environment through *Runtime.getRuntime()* and calling the *Runtime.exec(String command)* method, the Main class is able to launch an independent Java process and manage it - allowing the existance of the previously mentioned unique terminal for printing system status during runtime. The `command` string is built by retrieving the directory where the code is located:

```
java -cp <userdir>/build/classes fi.FarmInfrastructure
```


Main class launches Control Center and runs process for Farm Infrastructure
Both Control Center and Farm Infrastructure extend from javax.swing.JFrame
MessageProcessor and SocketServer implement Runnable
Farmer extends from Thread
EndSimulationException and StopHarvestException extend from Exception
and are thrown by all monitors' methods
Main
+ main()
- runProcess(String command)
- printLines(String cmd, InputStream ins)
EndSimulationException
StopHarvestException
SocketClient
- DataOutputStream: out
- Socket: socket
+ send()
+ close()
Farm Infrastructure
- Thread: serverThread
- SocketServer: fiServer
- SocketClient: ccClient
- MonitorMetadata: metadata
+ Integer: teamSize
+ Integer: pathSize
+ Integer: maxDelay
- Storehouse: storeHouse
- Standing: standing
- Path: path
- Granary: granary
- Farmer[]: farmerTeam
- CCProxy: messageProcessor
- (additional UI elements)
+ main()
- initComponents()
+ closeSocketClient()
+ close()
+ sendMessage()
- groupTextFields()
+ presentFarmerInStorehouse()
- clearStorehouse()
+ presentFarmerInStandingArea()
- clearStandingArea()
+ presentFarmerInPath()
- clearPath()
+ presentFarmerInGranary()
- clearGranary()
+ presentCollectingFarmer()
- clearCollectionArea()
- clearAll()
+ updateGranaryCornCobs()
+ updateStorehouseCornCobs()
Farmer
- FarmerState: state
- StorehouseFarmerInt: storeHous
- StandingFarmerInt: standing
- PathFarmerInt: path- GranaryFa
- Integer: id
- Integer: cornCobs
+ run()
+ getID()
+ getCornCobs()
+ setCornCobs()
<<Enumerate>>
FarmerState
INITIAL
PREPARE
WALK
WAITTOCOLLECT
COLLECT
WAITTORETURN
RETURN
STORE
<<Interface>>
Runnable
Control Center
- Thread: serverThread
- SocketServer: ccServer
- SocketClient: fiClient
- (additional UI elements)
+ main()
+ initFIClient()
+ closeSocketClient()
+ close()
+ enablePrepareBtn()
+ enableStartBtn()
+ enableCollectBtn()
+ enableReturnBtn()
- prepareBtnActionPerformed()
- startBtnActionPerformed()
- collectBtnActionPerformed()
- returnBtnActionPerformed()
- stopBtnActionPerformed()
- exitBtnActionPerformed()
MonitorMetadata
+ Integer: MAXNUMBERFARMERS
+ Integer: MAXDELAY
+ Integer: NUMBERFARMERS
+ Integer: NUMBERCORNCOBS
+ Integer: NUMBERSTEPS
+ Integer: TIMEOUT
<<Interface>>
StorehouseFarmerInt
+ farmerEnter(Integer farmerId)
+ farmerWaitPrepareOrder(Integer farmerId)
<<Interface>>
StorehouseCCInt
+ control(String action)
+ sendSelectionAndPrepareOrder
(Integer numberOfFarmers,
Integer numberOfCornCobs,
Integer maxNumberOfSteps,
Integer timeout)
+ waitAllFarmersReady()
Storehouse
- FarmInfrastructure: fi
- MonitorMetadata: metadata
- ReentrantLock: rl
- Condition: allInStorehouse
- Condition: prepareOrder
- Map<Integer, Integer>: positions
- List<Integer>: availablePosition
- Integer: farmersInStorehouse
- Integer: farmersSelected
- Integer: entitiesToStop
- Integer: cornCobs
- boolean: prepareOrderGiven
- boolean: stopHarvest
- boolean: endSimulation
- boolean: proxyInMonitor
+ farmerEnter(Integer farmerId)
+ farmerWaitPrepareOrder(Integer farmerId)
+ control(String action)
+ sendSelectionAndPrepareOrder (Integer
numberOfFarmers, Integer numberOfCornCobs,
Integer maxNumberOfSteps, Integer timeout)
+ waitAllFarmersReady()
- selectSpot(Integer farmerId)
- waitRandomDelay()
<<Interface>>
StandingFarmerInt
+ farmerEnter(Integer farmerId)
+ farmerWaitStartOrder(Integer farmerId)
<<Interface>>
StandingCCInt
+ control(String action)
+ sendStartOrder()
+ waitForAllFarmers()
Standing Area
- FarmInfrastructure: fi
- MonitorMetadata: metadata
- Map<Integer, Integer>: positions
- List<Integer>: availablePosition
- Integer: farmersInStanding
- Integer: entitiesToStop
- Integer: cornCobs
- boolean: startOrderGiven
- boolean: stopHarvest
- boolean: endSimulation
- boolean: proxyInMonitor
+ farmerEnter(Integer farmerId)
+ farmerWaitStartOrder(Integer farmerId)
+ control(String action)
+ sendStartOrder()
+ waitForAllFarmers()
- selectSpot(Integer farmerId)
- waitRandomDelay()
<<Interface>>
PathFarmerInt
+ farmerEnter(Integer farmerId)
+ farmerGoToGranary(Integer farmerId)
+ farmerReturn(Integer farmerId)
+ farmerGoToStorehouse(Integer farmerId)
<<Interface>>
PathCCInt
+ control(String action)
ConditionAndPathDepth
+ Condition: condition
+ Integer: position
+ Integer: depth
*
*
contains
Path
- FarmInfrastructure: fi
- MonitorMetadata: metadata
- ReentrantLock: rl
- Condition: allInPath
- List<Integer>: farmersOrder
- Map<Integer, ConditionAndPathDepth>:
farmersMetadata
- List<List<Integer>> availablePositions
- Integer[][]: path
- Integer: farmersInPath
- Integer: currentFarmerToMove
- Integer: entitiesToStop
- boolean: prepareOrderGiven
- boolean: stopHarvest
- boolean: endSimulation
- boolean: proxyInMonitor
+ farmerEnter(Integer farmerId)
+ farmerGoToGranary(Integer farmerId)
+ farmerReturn(Integer farmerId)
+ farmerGoToStorehouse(Integer farmerId)
+ control(String action)- selectSpot(Integer farme
- waitRandomDelay()
- waitTimeout()
<<Interface>>
UiAndMainControlsCC
+ presentFarmerInStorehouse(int farmerId, int pos)
+ presentFarmerInStandingArea(int farmerId, int pos)
+ presentFarmerInPath(int farmerId, int pos, int col)
+ presentFarmerInGranary(int farmerId, int pos)
+ presentCollectingFarmer(int farmerId)
+ updateGranaryCornCobs(int actualNumber)
+ updateStorehouseCornCobs(int actualNumber)
+ initFIClient()
+ enableStartBtn()
+ enableCollectBtn()
+ enableReturnBtn()
+ enablePrepareBtn()
+ closeSocketClientAndUI()
CCMessageProcessor
- UiAndMainControlsCC: cc
+ processMessage(String message)
SocketServer
- Integer: port
- MessageProcessor: mp
+ run()
<<Interface>>
MessageProcessor
+ processMessage(String message)
<<Interface>>
UiAndMainControlsFI
+ presentFarmerInStorehouse(int farmerId, int position)
+ presentFarmerInStandingArea(int farmerId, int position)
+ presentFarmerInPath(int farmerId, int position, int column)
+ presentFarmerInGranary(int farmerId, int position)
+ presentCollectingFarmer(int farmerId)
+ updateGranaryCornCobs(int actualNumber)
+ updateStorehouseCornCobs(int actualNumber)
+ sendMessage(String message)
+ closeSocketClientAndUI()
CCProxy
- UiAndMainControlsFI: fi
- StandingCCInt: standing
- PathCCInt: path
- GranaryCCInt: granary
- StorehouseCCInt: storeHouse
- String: message
+ processMessage(String message)
<<Interface>>
GranaryFarmerInt
+ farmerEnter(Integer farmerId)
+ farmerWaitCollectOrder(Integer farmerId)
+ farmerCollect(Integer farmerId)
+ farmerWaitReturnOrder(Integer farmerId)
<<Interface>>
GranaryCCInt
+ control(String action)
+ waitAllFarmersReadyToCollect()
+ sendCollectOrder()
+ waitAllFarmersCollect()
+ sendReturnOrder()
<<Interface>>
Runnable
ProcessingThread
-String: message
+ run()
Granary
- FarmInfrastructure: fi
- MonitorMetadata: metadata
- ReentrantLock: rl
- Condition: allInGranary
- Condition: allCollected
- Condition: waitCollectOrder
- Condition: waitReturnOrder
- Map<Integer, Integer>: positions
- List<Integer>: availablePosition
- Integer: farmersInGranary
- Integer: farmersCollected
- Integer: entitiesToStop
- Integer: maxCornCobs
- boolean: readyToCollect
- boolean: readyToReturn
- boolean: endSimulation
- boolean: proxyInMonitor
- boolean: stopHarvest
+ farmerEnter(Integer farmerId)
+ farmerWaitCollectOrder(Integer farmerId)
+ farmerCollect(Integer farmerId)
+ farmerWaitReturnOrder(Integer farmerId)
+ control(String action)
+ waitAllFarmersReadyToCollect()
+ sendCollectOrder()
+ waitAllFarmersCollect()
+ sendReturnOrder()
- selectSpot(Integer farmerId)
- waitRandomDelay()
- waitTimeout()
Figure 2: Class diagram of the Farm Simulation application.

Now we have our two entities - the system's core. ControlCenter and FarmInfrastructure implement the UiAndMainControlsCC interface and UiAndMainControlsFI interface respectively. Both classes extend from `javax.swing.JFrame` and contain the code regarding the GUIs and communicate with each other with the use of sockets. Both have instances of SocketServer and SocketClient, that in return have instances of `java.net.Socket`, and work as servers and clients simultaneously. Communication establishment is explained in section 4.2.

Farmers are simulated as thread instances from the same class. Their states are defined in the FarmerState enumerate. When instantiated by FI, they wait for orders that will start their lifecycle of retrieving and storing virtual corn cobs. MonitorMetadata, as the name states, contains the metadata of the monitors (the farm areas). This information applies constraints to the farm areas where farmers will pass, hence influencing the execution flow according to the users' wishes. As each farmer has its own execution thread, their access to shared resources must be controlled. This is where the monitors come in hand. The concurrency measures implemented by our monitors are justified in section 4.1.

3.2 User Interface

Having succinctly gone through all components, we move on to presenting the user interface and explaining its confinements. Built with Swing (*3*), the UIs remain faithful to the design concept proposed for the simulation, with some improvements.

In Figure 3 we see a screen capture during runtime of the FI interface. Here, the user holds no control and only sees what is happening to each Farmer thread by tracking the movement of their IDs. Note that the IDs correspond to internal class identifiers and not the actual thread identifiers. This means that, if desired, such IDs could be replaced by strings containing the farmer names for example (as long as they were unique, of course).

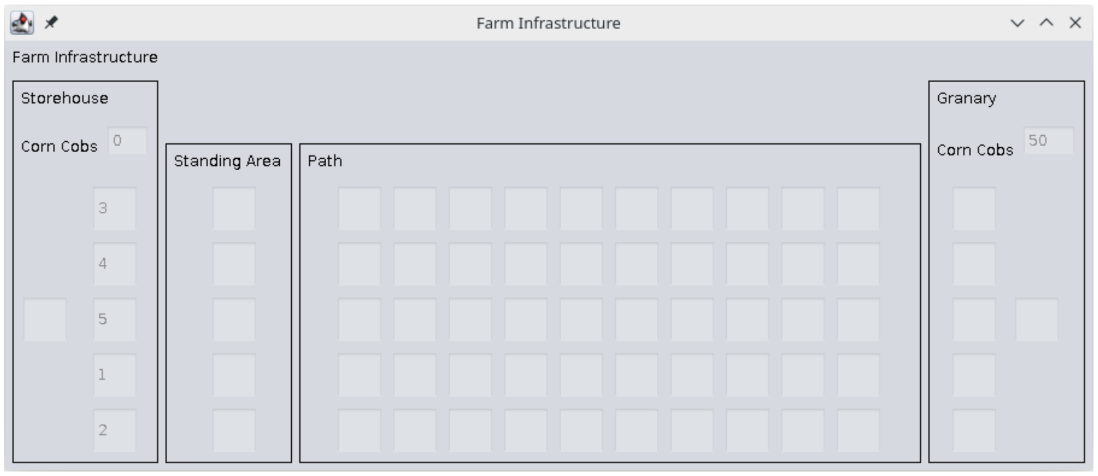
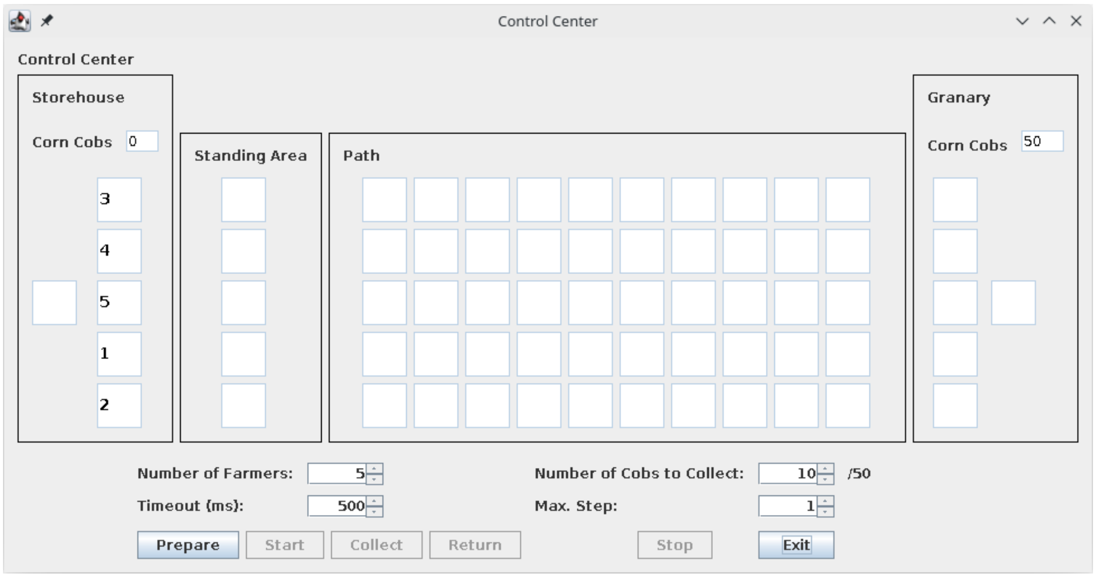


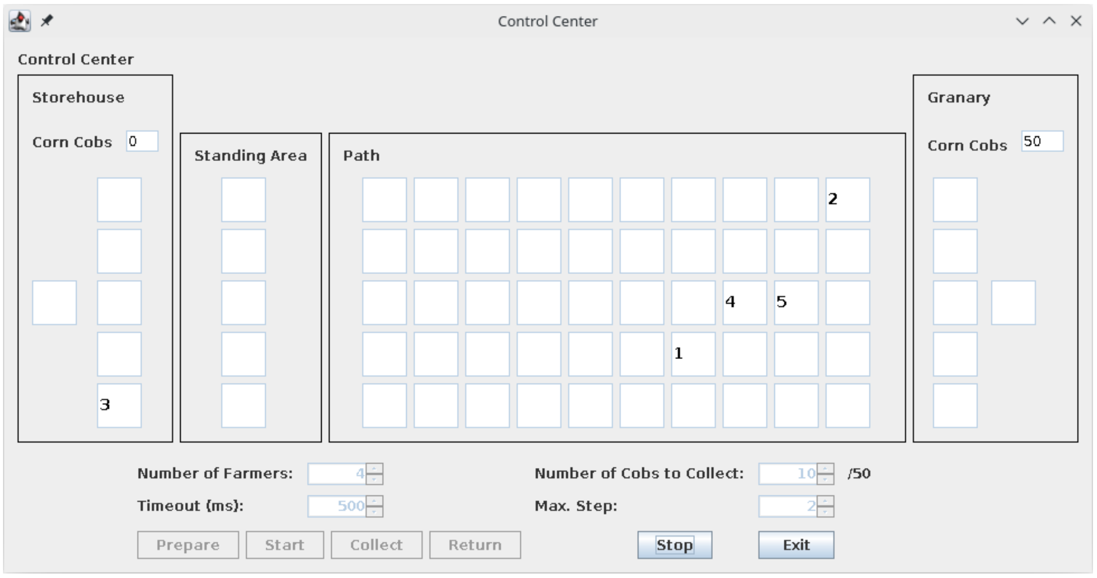
Figure 3: Screenshot of the FI UI.

Now Figures 4, 5 and 6 offer a more interesting view on the system. ....



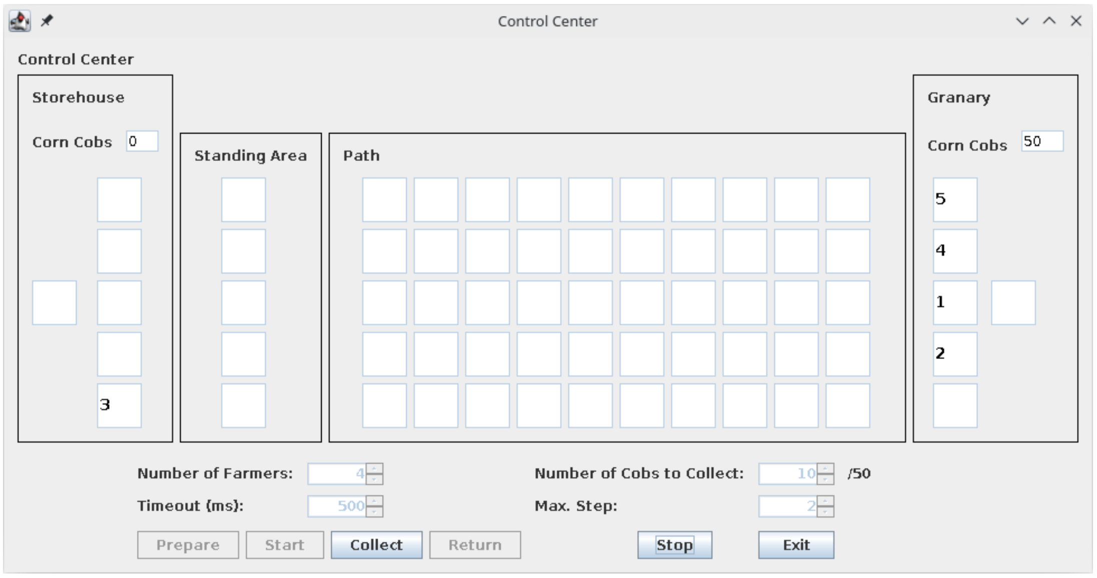
Storehouse Corn Cobs: 0
Granary Corn Cobs: 50
Number of Farmers: 5
Timeout (ms): 500
Number of Cobs to Collect: 10 /50
Max. Step: 1

Figure 4: Screenshot of the CC UI right after launching the application.



Storehouse Corn Cobs: 0
Granary Corn Cobs: 50
Number of Farmers: 4
Timeout (ms): 500
Number of Cobs to Collect: 10 /50
Max. Step: 2

Figure 5: Screenshot of the CC UI while farmers are crossing the Path.



Number of Farmers: 4

Timeout (ms): 500

Number of Cobs to Collect: 10 /50

Max. Step: 2

Figure 6: Screenshot of the CC UI when the farmers are ready to collect corn cobs.

4 Concurrency Strategy

.....

4.1 Monitors & Farmers

....

MonitorMetadata and 4 Monitors (and respective interfaces) Farmer and FarmerState

4.2 Communications

In this project, the communication system adopted and constrained by the guidelines provided was through sockets, using strings as messages. To enable this, we implemented two auxiliary classes to enable both client(*SocketClient*) and server(*SocketServer*) ready to use options. For each instance of SocketServer, a *MessageProcessor* instance must be provided as the internal message processing logic, and since in this problem there is two entities that need to receive and process messages, *CCMessageProcessor* and *CCProxy* were created. Taking a closer look at the communication system implemented, starting at the *SocketClient*, this class establishes a connection to the endpoint and port defined in its constructor, providing then a send method, which as the name suggest, sends a message to the socket server present in the other end of the connection and finally a close method, which as the name also implies, closes the socket connection previously established. In relation to the *SocketServer* class, as it suggests, it implements a ready to use server using sockets and deployed in the port provided in the constructor. It is implemented based on a thread, so that the server life-cycle can occur parallel to the main logic of the problem. The server is constantly waiting to receive a message, that when received is automatically passed to the MessageProcessor instance provided also in the constructor. The server will die after receiving and processing a previously defined message, in this case the string ”endSimulationOrder”.

As already mentioned, the *SocketServer* needs a *MessageProcessor* instance as its message processing logic. *MessageProcessor* is an interface that any class containing message processing logic should implement and that requires the implementation of the method `processMessage`. *CCMessageProcessor* implements the *MessageProcessor* interface and is responsible for processing any messages the Control Center receives. This implementation processes the messages in a sequential form, given the fact that the Control Center is mainly a GUI that enables the user to send commands to the Farm Infrastructure and visualize the state and positions of the farmers in the simulated farm, functionality added by our group since we consider the Control Center process and the Farm Infrastructure one independent although connected, and for that reason the processes can run in different machines, needing to know only the endpoints and ports, meaning that the Control Center, being the main entity of the simulation, should have also access to the state of the Farm Infrastructure and its components and workers.

On the other hand, the *CCProxy* acts as the message processor for the messages destined to the Farm Infrastructure as well as a substitute for the COntrol Center in the Farm Infrastructure process. Being a entity in a highly concurrent environment, a thread based processing was implemented, meaning that for each received message, a new `Thread` was initiated with its life-cycle defined in the `Runnable` *ProcessingThread*. This approach allows the correct handling of several messages at the same time, a crucial aspect in the Farm Infrastructure environment. The *ProcessingThread* class is the one containing the actual logic for the processing of each message the *FarmInfrastructure* receives.

4.3 Exceptions

When developing the solution for this complex concurrent scenario, the stop harvest and end simulation events were the ones who generated the most questions and difficulties. Since all threads can be locked in quite different points of its life-cycle, a solution to notify them all that doesn't implied verifying at each new step if in fact the harvest had stopped or the simulation had ended was not natural not immediate.

After many discussions and trials, the exception approach was selected and implemented. Exceptions enabled us to only verify if the harvest stopping and simulation ended conditions were true in a few strategic places, as well as in the case one of the conditions was verified, the corresponding exception was thrown and the current thread executing the code would automatically know, stop what they were doing and jump to the exception catching code, enabling us to process that event.

For this purpose, we create two custom exception classes, the StopHarvestSimulation and the EndSimulationException, both with self-explaining names.

5 Additional Remarks

5.1 Documentation

Packages, interfaces, classes and methods are all documented, with the help of Javadoc (*4*), and naming conventions are applied throughout the code, focusing on allowing readers to understand what methods do and variables are for by reading their names alone.

A significant effort was placed on this as we understood it would be greatly beneficial in the long term. And it was indeed. As soon as one developer completed a component and ensured the textual guidance, the other would quickly be up-to-date on the internal functioning of such component in his own tasks.

Creating the diagrams here presented also allowed to keep a wider perspective on the project and discuss important issues during debugging while maintaining the same mental picture.

5.2 Assignment Contributions

Regarding the work distribution amongst developers, a close-contact strategy was defined where each worked on a piece of software according to a predefined plan. The project structure and architecture was decided in conjunction, as well as the key concurrency solutions chosen. Both servers were also implemented collectively.

Nevertheless, some relatively independent task distribution was defined: João implemented the Granary and Path monitors, while Filipe did the Storehouse and Standing monitors; João established socket communications and respective message processors, while Filipe designed the user interface and respective interaction with the remaining components. Bug and error solving was made along the development phase by both developers any time it was required.

Once the final version of the application was completed, this report and the code documentation became our primary concern, with both contributing equally.

6 Conclusions

After completing the assignment, we drew a few conclusions regarding the topics here explored and our endeavor to deliver work of quality.

.....

References

1. Óscar Pereira, *SA: Practical Assignment no.1*, University of Aveiro, 2019/20.
2. Object Management Group, *What is UML*, `https://www.uml.org/what-is-uml.htm`, accessed in March 2020.
3. Oracle, *Swing*, `https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html`, accessed in February 2020.
4. Oracle, *Javadoc Technology*, `https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html`, accessed in February 2020.