Multi-Agent Systems

Supply Chain Simulation

Jostein Dyrseth - 40223535

December 7, 2018

1 Introduction

The scenario given, is about simulating a supply chain from the customers to the manufacturer and further to the suppliers. The main goal is to implement a working system, that calculates the profit of the manufacturer agent. The second goal is to experiment with the system to maximize the profit of the manufacturer. Other goals are to design a good ontology so that the agents can communicate and understand each other using the FIPA semantic language to pass messages in between each other.

In a multi-agent system, agents can be utilized to achieve better result than if humans were to manually do all the work. Agents are in many cases quicker, have more narrow intelligence, more ubiquitous and often better at delegating tasks.

Agents are representing different roles in real life. In this system there are three agent types. Agents representing the customers, an agent representing the manufacturer and supplier agents.

In object-oriented programming data are thrown around often in a closed environment. Objects call on methods and functions to achieve a behaviour, whereas in a multi-agent system agents are free to chose if they want to respond to incoming requests or not, or maybe they chooses to wait as there are other tasks that have a higher priority at the moment. Agents automate this process and are always alert (cyclic) to any updates, and

can therefore often achieve a better result than if humans were to monitor changes in the environment and manually taking planned actions.

Distributed agents can run overnight processing information and hopefully find their best solution, or at least a good enough solution. If cooperativeness is desirable, the agents can cooperate and achieve a global optima within the system. They can also be connected to a wast number of other agents, hence being able to join any environment no matter the scale. The only bottleneck at this point is computation.

2 Model Design

In this project, three agent types have been implemented to represent the customers, manufacturer and suppliers. There could have been more agent types like accountant and a warehouse agent, but this would make the system more complex so this was discarded. See Appendix 2 for more detail in the communication protocols.

2.1 Customer Agent

The customer agent is representing the potential customers that enroll into the system. Each customer generates and places one order for each day in the simulation. The order have been implemented as a concept as seen in Appendix 1. This will be further explained later. The communication protocol for the customer is an REQUEST performative. It is also listening for incoming INFORMS to receive it's delivery. It is also listening for a CONFIRM or REFUSE immediately back from the manufacturer to represent the receipt.

2.2 Manufacturer Agent

The manufacturer is the in-between agent that have to both please the customer as much as possible (listen for REQUESTS), but also to deal with buying components from the three suppliers depending on the due date. It also needs to send an CONFIRM or REFUSE back to the customer (receipt). Then each day listening to the supplier agents that give incoming supply-deliveries. It is also calculating the profit for each day as well

as adding up the total profit as will be further explained. Then it sends back to the original customer the PC.

2.3 Supplier Agent

The suppliers is simply listening for REQUESTS (cyclic) from the manufacturer. It will the sand back an ACL INFOROM message with the delivery concept in a supply-action. The role of this agent is simply to immediately react to the manufacturer and add the delivery in a future deliveries list.

2.4 Ontology

There are three ontology types. *AgentAction*, *Concept* and *Predicate*. In the implementation so far, only AgentAction and Concept have been used so far.

2.4.1 Components

The Comp class is an abstract concept. The components themselves extends the Comp Class. So for instance the Laptop_CPU component class extends the abstract CPU class which again extends the abstract Comp class. See Appendix 1. There are in total 11 non-abstract component classes and they don't hold any fields.

2.4.2 PC

The *PC* class is also a concept, therefore the implements "*Concept*" was chosen for this ontology class. It was decided that since this class can be of type *Laptop* and *Desktop*, it should be of an abstract class. It does not hold any fields, except two abstract function-declarations. The Laptop_PC for example is a non-abstract class which holds a list of components.

2.4.3 Order and Delivery

The *PC* is generated using the specifications given in the task. The *Order* gets generated in the *GenerateOrder* Behaviour. The *Order* is instantiated and then depending on the

random decimal value created between 0 and 1, it will set the *quantity*, *price*, *total_price* and the *due_in_days* values.

An Order is also a concept, so again the "Concept" implementation was chosen. The order also holds a *buyer*, along with the generated *PC*. Again see Appendix 1.

The Delivery (Concept) is very similar to the Order. It holds the current PC as well as deliver_in_days and total_cost values.

2.4.4 Buy and Supply

The Buy and Supply classes are actions. The Buy only holds an Order, while the Supply holds a Delivery and a receiver AID. The supply is used from the supplyer to the manufacturer and the Buy is used from both the customer to the manufacturer and from the manufacturer to the supplier.

2.4.5 Other Ontologies

Other classes like ConfirmOrder, RejectOrder and DeliverOrder have also been used. These are not shown as they are very similar. These goes from the manufacturer to the customer.

3 Model Implementation

Every agent enrolling into the system will immediately be readied in the "setup()" method. Here each agent instantiates and registers the ontology language, prints a hello-world message to the console to verify the agent has enrolled successfully and registers with the yellow pages in the directory facilitator so that other agents can see their future potential requests and services. After this, we add the behaviours to a queue. Every agent starts with adding the "*StartBehaviour*"-behaviour. Inside this behaviour we can further add the custom behaviours for the particular agent. Since these features are present in all agents, there could have been implemented an abstract agent class to avoid redundant code int this case, however this decision was discarded as the code is not of any significant amount.

3.1 Customer

3.1.1 Customer Behaviours

The customer agent holds a set of behaviours that are both "*OneShotBehaviours*" or "*CyclicBehaviours*". Since the customer has many subtasks that needs to be taken care of in a sequence a *SequentialBehaviour* was implemented with the name "*dailyActivity*". This behaviour holds the current subtasks:

**FindManufacturer *GenerateOrder *SendOrder *EndDay*

When the agent wants to receive an order it is happening as a cyclic-behaviour: "*ReceiveOrder*". This was the most logical approach as the receiver is in a waiting state, and does not necessarily know when the order is to arrive. It also has a cyclic behaviour to receive the receipts.

3.1.2 Customer Constraints

When the customer is generating a PC in the *GenerateOrder* class, either a Laptop- or a Desktop PC will be instantiated. This is dependant on some randomly created decimal values between 0 and 1. The Laptop and Desktop will always have certain constant fields, so it was clear that these two sub-classes would have a constructor setting these fields to the list of components immediately. For instance in the Laptop concept-class, the constructor immediately adds three components to it's list of components: a laptop CPU, a laptop motherboard and a screen. See Listing 1.

After the instantiating of either a Laptop or a Desktop, the subsequent components will be set. This also happens randomly the same way as described above. After all components have been set, the Order can add the PC as well as randomly setting some further values.

3.2 Manufacturer

3.2.1 Manufacturer Behaviours

While the manufacturer agent has the same underlying concept as the customer - holding behaviours within a cyclic sequence behaviour - it also adds a cyclic behaviour (*ReceiveSupply*) alongside the sequence behaviour. This cyclic behaviour is running in the background until the agent sends a "new day" message to the *SynchDaysAgent*.

The manufacturer also holds: *FindSuppliers*-, *ReceiveAndForwardCustumerOrders*- and the *EndDay* Behaviour in a sequence. These are all *OneShotBehaviours*. The find suppliers and end day are quite self explanatory.

The *ReceiveAndForwardCustumerOrders* behaviour is calculating which suppliers to forward the received order to. This is done in the *calcBestSupplier* method as seen in Listing 2. Based on this value the order gets forwarded to it's associated supplier using a buy-on-demand strategy. The manufacturer can already now calculate the profit based on the cost from the supplier. The manufacturer know this already because it reads in an argument *sup_info* in the beginning. The main application class is reading in a .csv file with the cost for each component from each supplier (see Listing 3). The agent then uses this data to calculate the profit. It also sum up the profits for each order for each day to eventually output the total profit earned. If the profit of a single order is positive, then CONFIRM (accept order) the request from the customer, wrap the order in a buy *AgentAction-action* and forward it, else it will REFUSE the request.

3.2.2 Manufacturer Constraints

Both penalty for late delivered and per-component per-day warehouse storage cost was not being implemented. This is because of the manufacturer will always order the components in bulk and never order any single components, hence never have any components in it's warehouse overnight (it can manufacture and deliver on the same day).

3.3 Supplier

3.3.1 Supplier Behaviours

The supplier have behaviours *DeliverSupply* (one-shot sequence) and *ReceiveBuyRequest* (cyclic). The latter is listening for a *ACL REQUEST-message*. Processing incoming messages the supplier verifies that the Buy-action holds the appropriate ontology. After verifying the message, it will extract the order (concept), calculate the delivery day and wrap the content in a Supply action-class. This also holds a receiver field of the manufacturer's AID ready to be sent. But for now, the Supply object gets added to a list that holds all future supplies.

The *DeliverSupply* will simply look into the list of all the future supplies and see if any supplies are expected outbound on the current day. After the supply has been sent, it get's removed from the list to hold it tidy. This repeats for as many times as the *daily_list's* length is.

As with the Supplier constraints the component-delivery-time from the three suppliers are as described above.

4 Design of manufacturer agent control strategy

Since the control strategy used is using a "buy-on-demand" strategy, the penalty for late orders and warehouse costs can be discarded, making the system a bit less complex. This strategy has an advantage, but it also has a weakness as it is not able to hold anything in stock which could give potentially even quicker deliveries. The manufacturer decides which supplier to order from depending on the due_in_days value. See Listing in Appendix 3.

The manufacturer now needs to either CONFIRM or REFUSE the order from the customer (listing in Appendix 3). Much of this have already been discussed.

5 Experimental results

After running the simulation 5 times this table were produced giving an average profit £ 1 963 013:

1	£ 2 015 156
2	£ 1 986 273
3	£ 1 830 145
4	£ 2 281 619
5	£ 1 701 876

The only parameter that can be changed is the number of customers. This was changed to c=10.

1	£ 6 284 920
2	£ 5 836 130
3	£ 5 279 481
4	£ 7 205 716
5	£ 6 417 328

Again giving an average of £ 6 103 712. This is better profit. We can see that this system is easily scaleable. See Bar Chart in Appendix 2.

Other parameters were not possible to evaluate using this buy-on-demand strategy.

6 Conclusions

The customer could have sent a QUERY_IF FIPA-performative before doing the REQUEST to check whether it is in stock or not. The manufacturer could then PROPOSE to let the customer do that order or not following placing the order normally (REQUEST). The manufacturer does neither yet take the costs from the customer. This should have been implemented, but rather is taken care of internally in the manufacturer. The Predicate type should have been used as well, but it's not because of time constraints. This could have been used for informing what components belong to whom at each moment for instance.

Some strengths are: quickly orders supply, never holds anything in the warehouse, never delivers too late, hence needing to pay for per component per day costs. Also It will run quickly and not go into any deadlock states. As well as calculating the profit as specified in the requirements. It is also able to cope with as many customers as possible. both c=3 and c=10 were tested.

Some weaknesses are: that it is not able to predict, analyze or order components in advance (that are highly likely to be in a future order). This is not a great thing when it comes to achieving a greater profit and delivering faster to the customer.

However the simulation is a robust and a great beginning. If the project was of greater scale strategies like discussed above would indeed be implemented easily on top of this design.

7 References

No references.

8 Appendix 1: Ontology

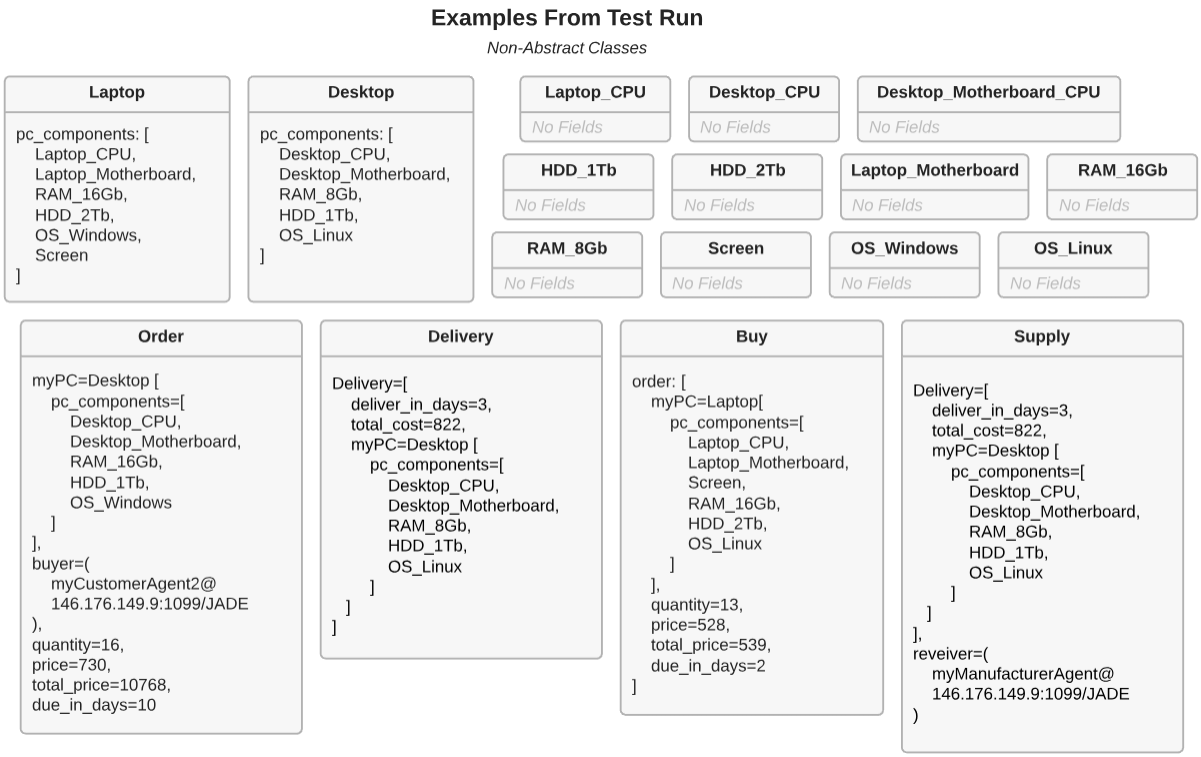

Figure 1: Example Instances for each non-abstract classes (Excluding the Agent Classes).

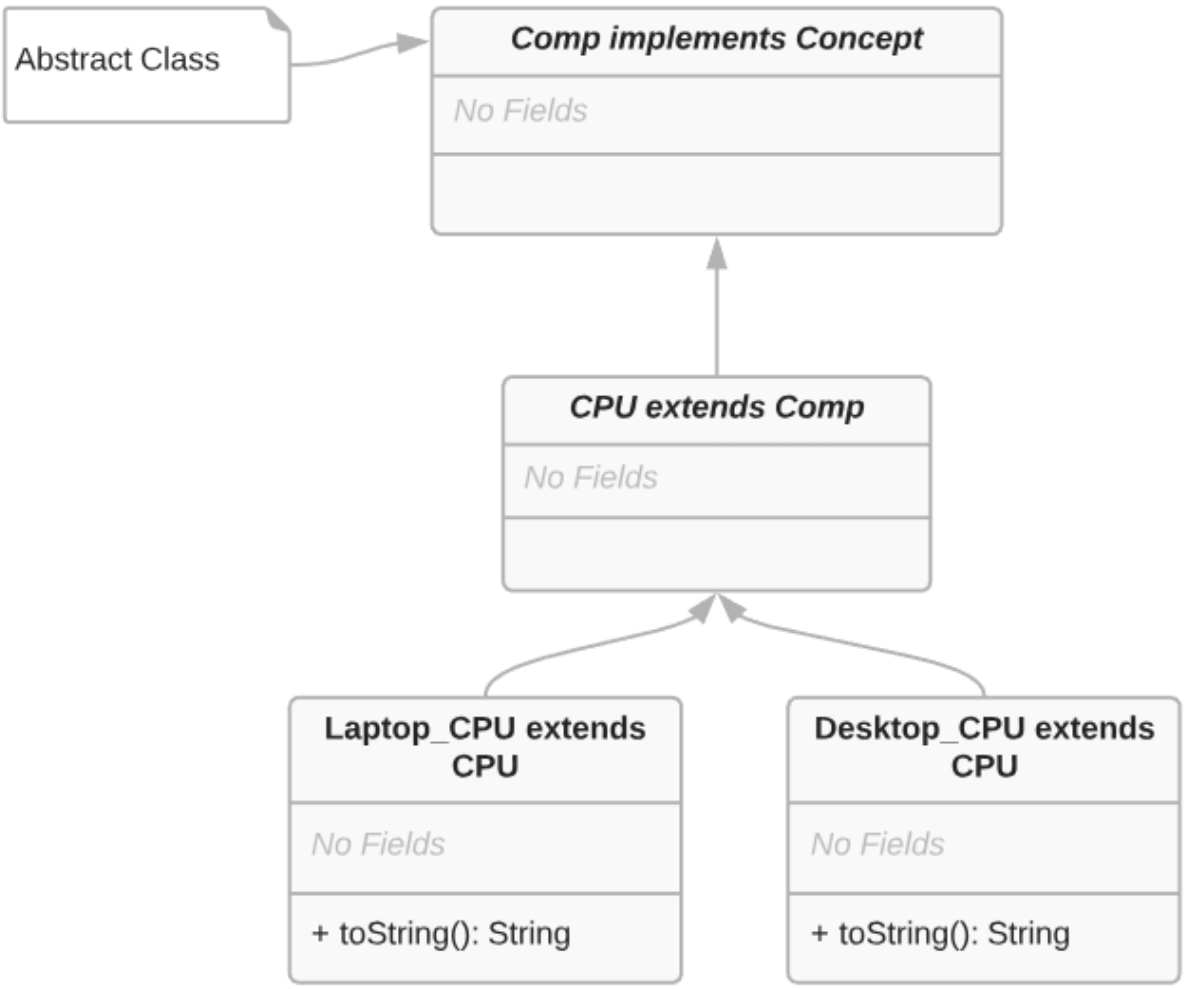

Figure 2: Comp Class

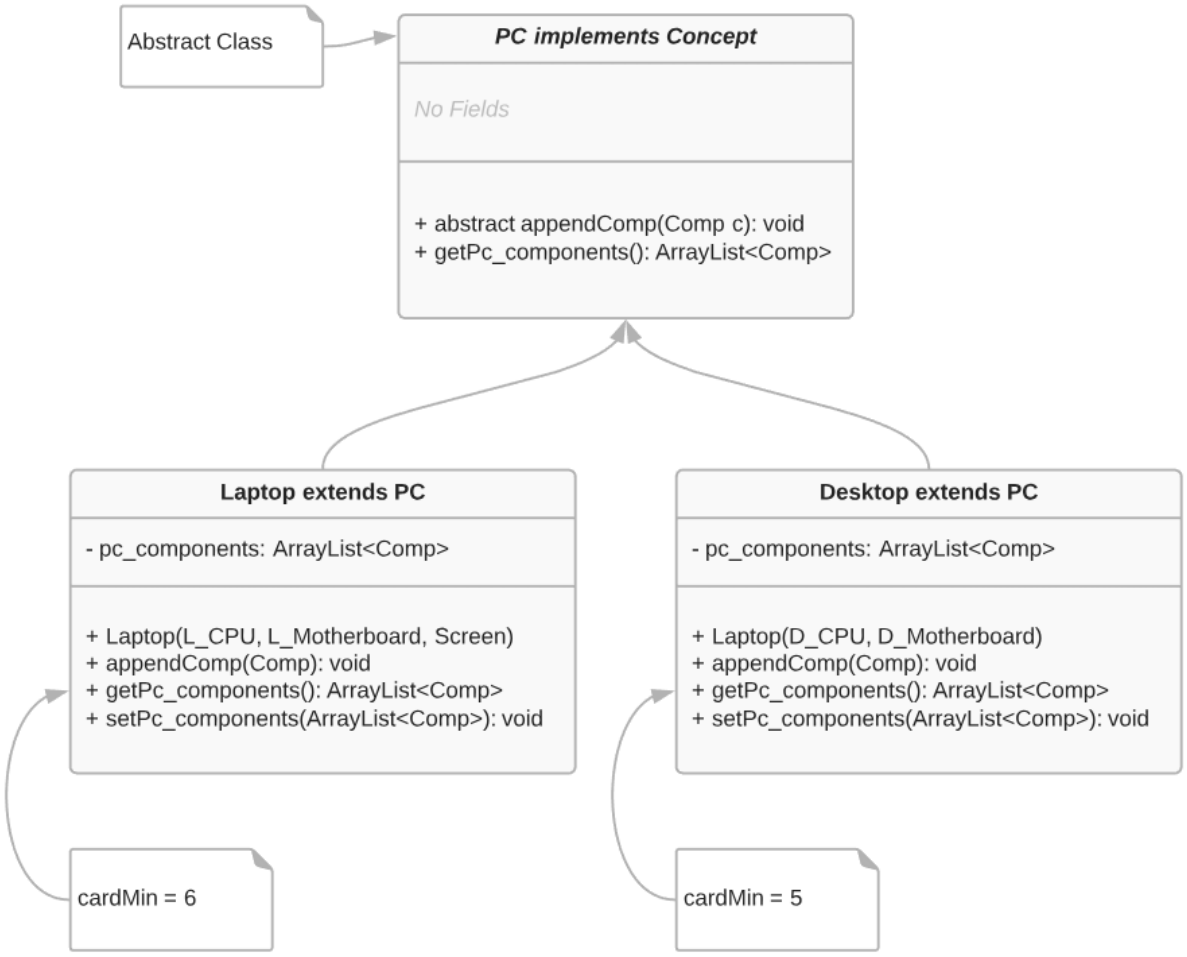

Figure 3: PC class

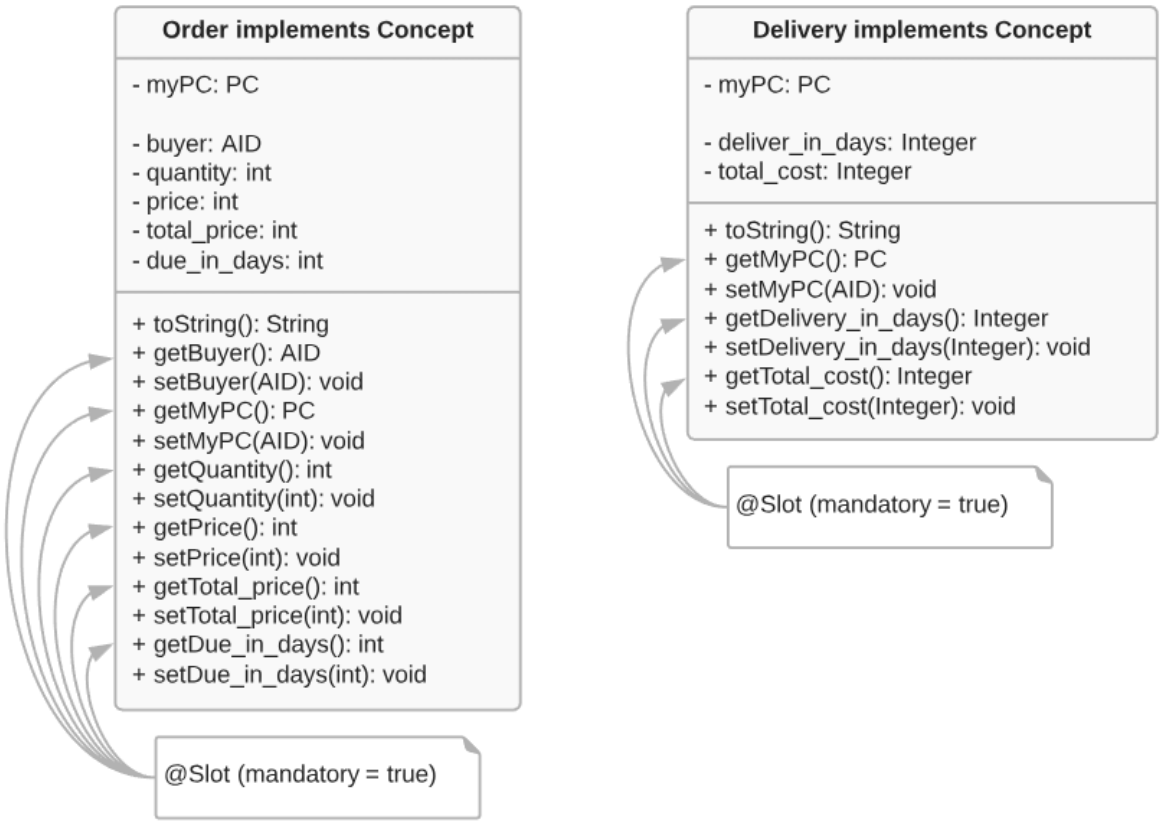

Figure 4: Order and Delivery Classes

Figure 5: AgentActions

9 Appendix 2: Communication Protocol

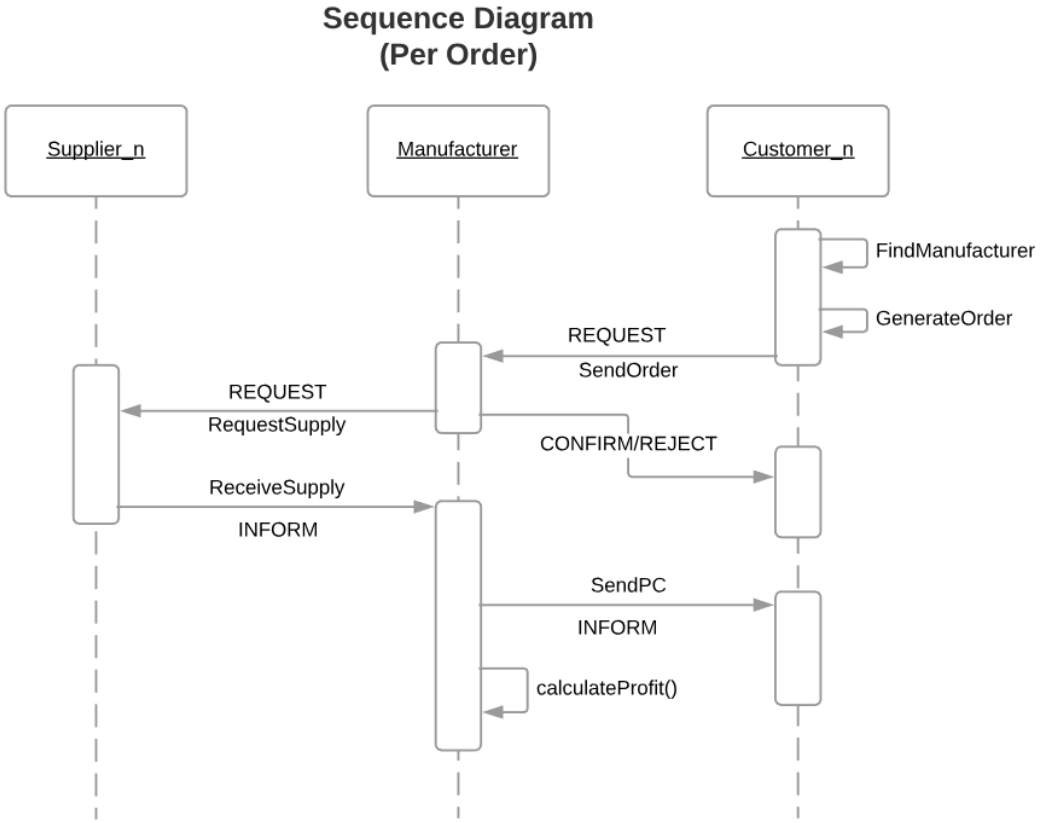

Figure 6: Sequence Diagram.

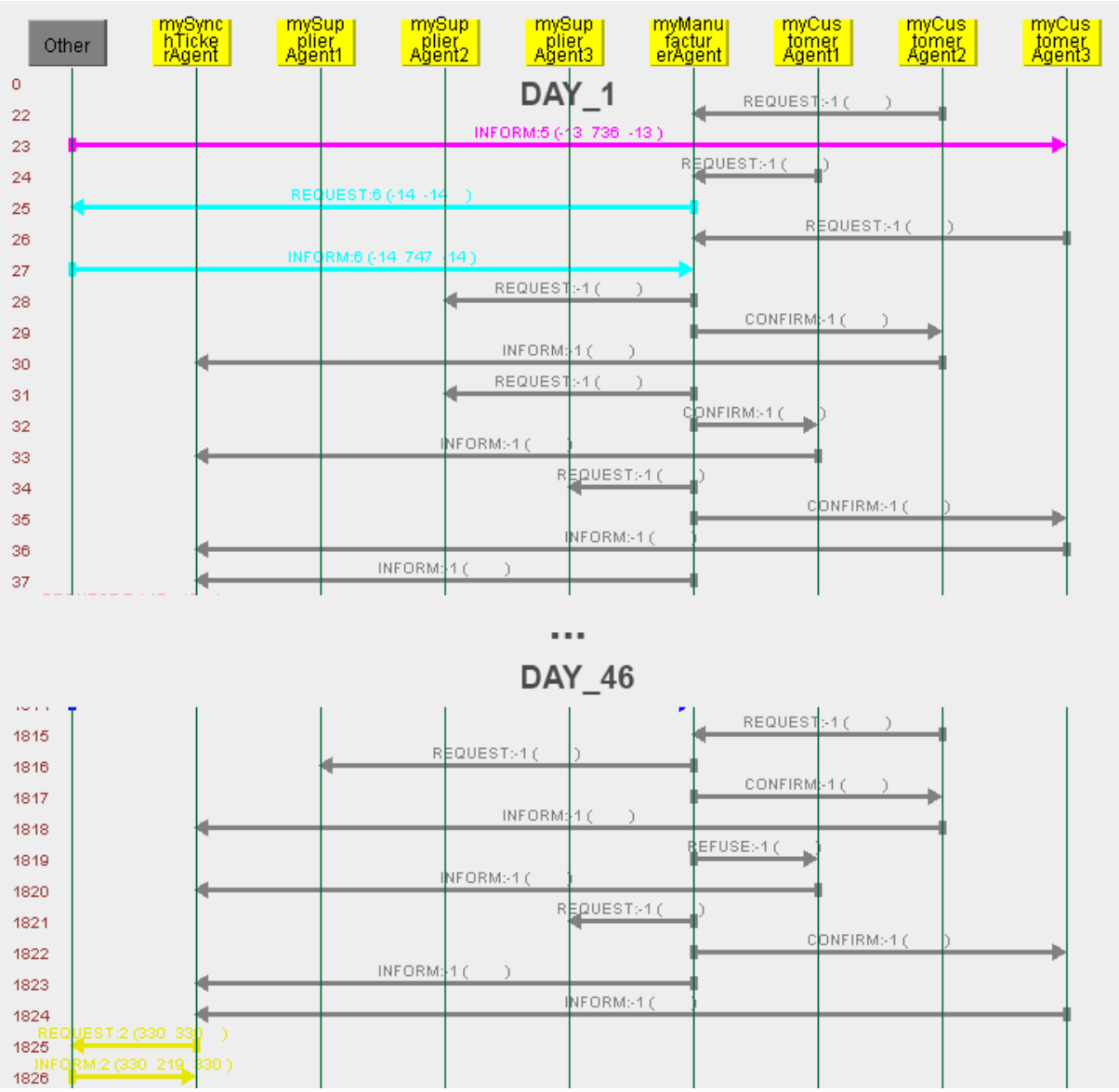

Figure 7: Sniffer screenshot 1: The first day all the messages between the agents and the SynchTickerAgent will take place, following by the performatives between the agents themselves.REFUSE sent on line 1819. This id day 46 in the simulation.

```
(INFORM
    :sender ( agent-identifier :name mySupplierAgent3@172.30.97.1:1099/
    JADE :addresses (sequence http:/ME0B056-01606B.c27-merch.napier.ac.uk
    :7778/acc ))
    :receiver (set ( agent-identifier :name myManufacturerAgent@172
    .30.97.1:1099/JADE :addresses (sequence http:/ME0B056-01606B.c27-merch.
    napier.ac.uk:7778/acc )) )
    :content
        ((action (agent-identifier :name myManufacturerAgent@172
    .30.97.1:1099/JADE :addresses (sequence http:/ME0B056-01606B.c27-merch.
    napier.ac.uk:7778/acc))
            (Supply :delivery (Delivery
                :myPC (Desktop :pc_components (sequence
                    (Desktop_CPU)
                    (Desktop_Motherboard)
                    (RAM_8Gb)
                    (HDD_2Tb)
                    (OS_Linux)))
                :receiver (agent-identifier :name myCustomerAgent3@172
    .30.97.1:1099/JADE
                :addresses (sequence http:/ME0B056-01606B.c27-merch.napier.
    ac.uk:7778/acc))))))
        :language fipa-sl :ontology my_ontology
)
```

Listing 1: Example Supply action content.

```
Content (
    action (
        agent-identifier
            :name mySupplierAgent2@172.30.97.1:1099/JADE
            :addresses (sequence http:/ME0B056...)

        Buy
        :order (
            Order
                :buyer (agent-identifier :name myCustomerAgent3@172.30...)
```

```
                :due_in_days 8
                :myPC (Desktop
                    :pc_components (sequence
                        (Desktop_CPU)
                        (Desktop_Motherboard)
                        (RAM_8Gb)
                        (HDD_1Tb)
                        (OS_Linux)
                    )
                )
                :price 737
                :quantity 9
                :total_price 7137
            )
        )
    )
)
```

Listing 2: Example Buy action content.

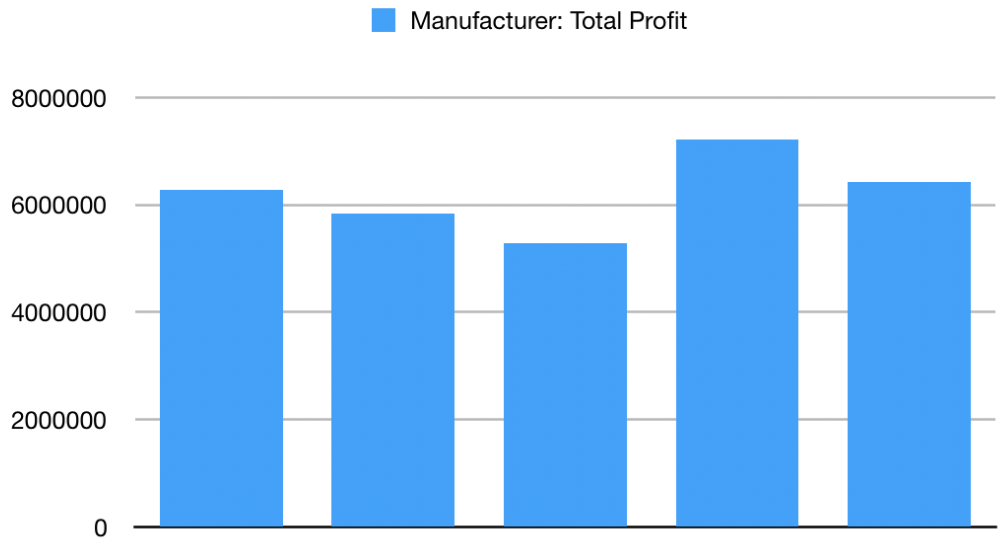

Figure 8: Table: Manufacturer Profit: 10 Customers.

10 Appendix 3: Source Code

```
public class Laptop extends PC {

  private ArrayList<Comp> pc_components = new ArrayList<Comp>();

  // Constructor
  public Laptop() {
    this.appendComp(new Laptop_CPU());
    this.appendComp(new Laptop_Motherboard());
    this.appendComp(new Screen());
  }
  // Add a single component
  public void appendComp(Comp c) {
    this.pc_components.add(c);
  }
  // toString
  @Override
  public String toString() {
    return "Laptop [pc_components=" + pc_components + "]";
  }
  @AggregateSlot ( cardMin = 6)
  public ArrayList<Comp> getPc_components() {
    return pc_components;
  }
  public void setPc_components(ArrayList<Comp> pc_components) {
    this.pc_components = pc_components;
  }
}
```

Listing 3: The constructor adds three standard Laptop components to the component list.

```
// Read CSV File
String csvFile = "src/sup_data/sup_data.csv";
BufferedReader br = null;
String line = "";
```

```
String cvsSplitBy = ",";

...

AgentController ManufacturerAgent = myContainer.createNewAgent("
    myManufacturerAgent", ManufacturerAgent.class.getCanonicalName(),
    suppliers_info);
ManufacturerAgent.start();
```

Listing 4: Reading in data from the CSV file and writes it to the agent's argument-input.

```
public Integer calcBestSupplier(Order order):
  if: order.due_in_days >= 7 // supplier3
    return 3;
  else if: order.due_in_days >= 3 // supplier2
    return 2;
  else: // supplier1
    return 1;
```

Listing 5: Pseudocode of how to find the correct supplier.

```
if profit_on_single_order is positive:
  if components are in stock:
    print: "components are in stock"(This will not happen in Buy-on-demand)

  else components are not in stock:
    // forward order to supplier

    current_profit = profit_on_single_order * order.getQuantity()

    if day <= 90:
      total_profit += current_profit;
        send REQUEST to current_supplier (Buy-Action)
        send CONFIRM back to customer
else:
  send REFUSE back to customer
```

Listing 6: Pseudocode of whether to CONFIRM or REFUSE the customer Order.

```
private Integer calcMinCostFromSupOrder(current_supplier, order):
  current_order_cost = 0
  for each Comp comp : MyPC.pc_components:
    current_order_cost += comp.cost

  return current_order_cost;
```

Listing 7: Pseudocode of how to get the current order costs. We can now easily calculate the profit with subtracting the total cost from supplier from this value times the quantity.