Edinburgh Napier UNIVERSITY

Multi-Agent Systems - Coursework Report

Marcos Ramirez Aceves
40284094@live.napier.ac.uk
Edinburgh Napier University - Multi-Agent Systems (SET10111)

1 Introduction

The aim of this coursework is the creation and evaluation of a multi-agent system for the management of a smartphone supply chain, which can be seen as an example of a supply chain management system or SCM.

A supply chain can be described as a network of different components like suppliers, warehouses, manufacturers, factories, distribution centers, retailers and customers, through which raw materials are acquired, transformed and finally deliver to customers [4]. Traditionally, this would involve a long process of communication, cooperation, coordination and negotiation, that would have an effect in the final cost of the production. An approach to improve this process is the introduction of computer methods like multi-agent systems. Furthermore, the components or entities mentioned before can be seen as intelligent software agents which are responsible for certain activities in the supply chain and that need to collaborate in order for the final product to be assembled.

Regardless of the kind of business involved in SCM systems, the main objective of any business is to earn the maximum profit by increasing the working capital, streamlining accounts, managing debtors and eliminating unnecessary costs [3]. Therefore, a mechanism to handle the complex communication that takes place between all the components in a supply chain is crucial and is the reason why this kind of problems are important.

Nowadays, because of the world-wide competitive market in which many businesses are involved, the use of traditional methods to achieve a good level of communication and cooperation between the different elements involved in the supply chain is not enough. The use of multi-agent systems for SCM can help the coordination between the different parts involved in the supply chain resulting in a faster and better cooperation between them. Multi-agent systems can also help in automating the buyer-seller negotiations and provide aid in the decision making process. In addition, multi-agent systems can be very useful in situations in which an unexpected rush of orders cause delays and decreases efficiency, as intelligent agents need to be capable of adapting to different situations and taking response accordingly, something that would be much more time and resource consuming if traditional methods were used. Other advantages that multi-agent systems can provide to SCM are: increase the supply chain efficiency, improve flow of materials between the suppliers and the manufacturer or reduce the total cost of the process [5].

2 Model Design

In this coursework, the following five agents will be used for the implementation of a multi-agent system for the management of a smartphone supply chain: CustomerAgents, ManufacturerAgent, SupplierAgents, WarehouseAgent, DayCoordinatorAgent. These agents will have the following roles in the system:

- CustomerAgents are responsible of:
  - Request orders to the ManufacturerAgent.
  - Send payment of orders to the ManufacturerAgent after receiving them.
- ManufacturerAgent is responsible of:
  - Decide whether or not to accept the orders requested by CustomerAgents.
  - Request the WarehouseAgent to calculate the costs of an order.
  - Request the WarehouseAgent to assemble orders.
  - Receive orders that are ready from the WarehouseAgent to then send them to the corresponding CustomerAgent.
  - Calculate daily and overall profit.
  - Call the day off.
- WarehouseAgent is responsible of:
  - Calculate costs of an order and send the result back to the ManufacturerAgent.
  - Process and assemble orders requested by the ManufacturerAgent.
  - Send ready orders to the ManufacturerAgent.
  - Decide which components order and how many from each SupplierAgent.
  - Receive components from SupplierAgents.
  - Maintain appropriate stock levels.
  - Calculate storage, penalties and supplies bought costs.
- SupplierAgents are responsible of:
  - Process requests of components sent by the WarehouseAgent.
  - Send components when ready to the WarehouseAgent.
- DayCoordinatorAgent is responsible of:
  - Inform all agents that a new day has begun.
  - Inform all agents that the simulation has finished.
  - Process day end calls from the ManufacturerAgent.

In relation to the ontology design, two kinds of smartphones are going to be produced in the supply chain: small smartphones and phablet smartphones. These two types of smartphones are going to have a specific type of screen and battery but variable storage and ram, for this reason, a base concept for an smartphone with the four properties is established in the ontology and from it, two sub-concepts are created, one for small smartphones and one for phablet smartphones, this can be seen in figure 5. In relation to the elements that the smartphones are going to be made of, each element is going to have a certain price and another property, therefore, as shown in figure 5, a base concept that only is going to have a price property called Component is settled in the ontology, the four different components that smartphones are made of inherit from it, these concepts being: Screen, Storage, Ram and Battery. Finally, one concept is created for the orders that the customers are going to request to the manufacturer and another concept is created for the supplies that the warehouse is going to receive from the suppliers.

Regarding the communication protocols chosen, for the communication between the customer and the manufacturer shown in figure 8, the only protocol used is a FIPA-Request protocol in which a customer is

going to request the manufacturer to sell an order to it. After doing so, the manufacturer will either agree or refuse to process the order. The request message has as content the agent action SellOrder, which can be seen in figure 7. In the case that the manufacturer accepted the order, later on, it will send an inform message with the predicate OrderDelivered as the content to the customer, which then will respond with another inform message with the predicate Payment as the content, both predicates can be seen in figure 6.

About the communication between the manufacturer and the warehouse shown in figure 9, three protocols are used. The first one is a FIPA-Request protocol in which the manufacturer is going to request the warehouse to calculate the cost of an order, this will be achieved by sending a request message with the agent-action CalculateOrderCost shown in figure 7 as the message content, after doing so, the warehouse will respond with an inform message with the predicate PredictedOrderCost, shown in figure 6, as the content. For this first protocol, there is no agree or refuse stage as the warehouse will always accept the request. The second protocol is another FIPA-Request protocol which cannot be refused by the warehouse, in this case, the manufacturer is going to request the warehouse to assemble an order using the agent-action PrepareOrderToAssemble as the content, as the order will not be assembled immediately, the warehouse will respond when the order is ready using an inform message with the OrdersReady predicate as its content. The third and final protocol used in the communication between these agents is a FIPA-Query protocol, in which the manufacturer will ask the warehouse about its daily expenses using a query message with the predicate WarehouseExpensesToday as the content of it, after that, the warehouse will respond with an inform message using the predicate WarehouseExpenses as the content, this message will carry the different expenses that will be used by the manufacturer to calculate the daily profit.

With reference to the communication between the warehouse and the supplier shown in figure 10, two protocols are used. The first one is a FIPA-Query protocol in which the warehouse is going to ask the supplier about its details, this will be carried out with a query message that has the SupplierDetails predicate shown in figure 6 as the content, and to which the supplier will respond with an inform message containing the predicate SupplierInformation. This interaction will provide the warehouse with the different components and prices that the supplier offers, as well as the delivery times of any orders requested to it. The second communication protocol is a FIPA-Request protocol in which the warehouse will request the supplier to sell some components to it, this will be accomplished through a request message with the agent-action SellSupplies shown in figure 7 as the content, and to which the suppplier will respond with an inform message that will have as content the predicate SuppliesDelivered. It can be seen in figure 10 that the inform message is before the request message, the reason for it is that the supplier will send the supplies days after the request, which means that, in each day, the warehouse will receive supplies requested from previous days and will request for new supplies that will arrive during the following days.

Finally, the communication between the DayCoordinator and the rest of the agents do not use any FIPA protocols as the only kind of messages exchange are inform messages that will keep the agents synchronized, this can be seen in figure 11.

3 Model Implementation

All agents are going to have two types of similar behaviours: finder behaviours and a synchronizing/waiter behaviour. The finder behaviours are one-shot behaviours that are just going to look for other agents in the yellow pages and the synchronizing/waiter behaviour is a cyclic behaviour that is going to handle the communication with the DayCoordinator, an example of this last one can be seen in listing 1. Apart from those two types, each agent implements the following behaviours:

- DayCoordinatorAgent behaviours:
  - $SyncAgentsBehaviour$: Cyclic behaviour that is going to keep all agents synchronized as

can be seen in listing 2. This will be achieved by sending to all agents a $NewDay$ predicate after receiving an $EndDay$ predicate from the manufacturer, this behaviour will also send a termination message when the simulation finishes using the $EndSimulation$ predicate.

- CustomerAgent behaviours:
  - $RequestOrderBehaviour$: One-shot behaviour which is just going to request a new order to the manufacturer everyday via the $SellOrder$ agent-action, shown in listing 3.
  - $ReceiveOrderBehaviour$: Cyclic behaviour that is going to receive ready orders from the manufacturer and send back a payment with the predicate $Payment$, shown in listing 4.
- SupplierAgent behaviours:
  - $ProvideDetailsBehaviour$: This behaviour is going to send to the warehouse its information via the $SupplierInformation$ predicate, shown in listing 5.
  - $ProcessSuppliesRequestsBehaviour$: Cyclic behaviour that is going to receive $SellSupplies$ agent-actions requests to send components and is going to prepare them to be delivered. This can be seen in listing 6.
  - $SendSuppliesBehaviour$: One-shot behaviour that is going to send the supplies to the warehouse when they are ready using the predicate $SuppliesDelivered$ as shown in listing 7.
- ManufacturerAgent behaviours:
  - $ProcessOrderBehaviour$: Multi-step behaviour, which can be seen in listing 8 that is going to receive $SellOrder$ agent-action requests from customers, request the warehouse to calculate a predicted cost using the agent-action $CalculateOrderCost$ and decide whether to process the order or not.
  - $ProcessOrdersReadyBehaviour$: Multi-step behaviour that is going to receive orders ready from the warehouse, send them to their corresponding customers using the predicate $OrderDelivered$ and receive payment for them, this is shown in listing 9.
  - $CalculateDailyProfitBehaviour$: Multi-Step behaviour that is going to query the warehouse for its daily expenses using the predicate $WarehouseExpensesToday$ and is going to calculate the daily profit as evidenced in listing 10.
  - $EndDayBehaviour$: One-shot behaviour that is going to call thed day off by send the predicate $DayEnd$ to the DayCoordinator, as demonstrated in listing 11
- WarehouseAgent behaviours:
  - $GetSupplierDetailsBehaviour$: Multi-step behaviour that queries the suppliers about their details using the predicate $SupplierDetails$ and process the response as seen in listing 12.
  - $ProcessSuppliesDeliveredBehaviour$: Behaviour that is going to receive the components from the suppliers via the $SuppliesDelivered$ predicate and place them in the stock as shown in listing 13.
  - $ProcessPendingOrdersBehaviour$:One-shot behaviour that is going to try to assemble orders that are pending.
  - $UpdatePendingOrdersTimesAndCalculatePenaltiesBehaviour$: One-shot behaviour that is going to update due dates and left days to assembly of orders and calculate daily penalties for late orders.
  - $ProcessManufacturerOrderRequestsBehaviour$: Behaviour that is going to predict the cost of an order and prepare orders to assemble when receiving the predicates

$CalculateOrderCost$ and $PrepareOrderToAssemble$ respectively. This can be seen in listing 14.

- $RequestSuppliesToSuppliersBehaviour$: One-shot behaviour that is going to request components to the supplier via the agent-action $SellSupplies$ as evidenced in listing 15.
- $SendReadyOrdersBehaviour$: One-shot behaviour, which can be seen in listing 16, that is going to send orders ready to the manufacturer using the $OrdersReady$ predicate.
- $CalculateStorageCostBehaviour$: One-shot behaviour that is going to calculate the costs of the components storage.
- $SendDailyWarehouseExpensesBehaviour$: Multi-step behaviour, shown in listing 17, that is going to send to the manufacturer the different warehouse expenses via the predicate $WarehouseExpenses$.

In relation to the implementation required constraints: as can be seen in listings [18,19,20,21], each component concept specific property can only have two types of values which are enforced by setting the slot permitted values to them, the ontology enforces the smartphones to have all the components by making them mandatory as shown in listing 22, and the screen and battery specifications for each type of smartphones are enforced by having their setters in the base class as protected and setting them in the constructor of the extended smartphone concepts as evidenced in listings [23,24].

The component delivery times are enforced by the supplier's $ProcessSuppliesRequestsBehaviour$, which is going to create an internal representation of the supplies to deliver and set the time to deliver them based on its own delivery time which is passed as an argument to the agent when it is created, and the $SendSuppliesBehaviour$ which makes sure that the supplies are only delivered when the days left to deliver reach zero, this is evidenced in listings [6,7].

The per-component per-day warehouse storage cost is enforced by the warehouse's $CalculateStorageCostBehaviour$, shown in listing 25, which is going to multiply the number of components in stock by the warehouse's property $perComponentDailyStorageCost$.

Shipping an order when there are enough components to build all the smartphones in it and making sure that a maximum of fifty smartphones are assembled on one day is going to be enforced by the warehouse's $ProcessPendingOrdersBehaviour$, which can be seen in listing 26, this behaviour is going to stop processing orders if the number of smartphones processed reaches the maximum, if the components needed are not in stock or if there are not enough components to assemble the order.

The penalties for late deliveries are calculated by the warehouse's $UpdatePendingOrdersTimesAndCalculatePenaltiesBehaviour$, shown in listing 27, which is going to decrease the due date of the pending orders and if it is below zero then is going to add the penalty to the $dailyPenaltiesCost$ property.

Finally, the calculation of the profit at the end of each day is done by the manufacturer's $CalculateDailyProfitBehaviour$, which is going to request to the warehouse about its daily expenses and it is going to extract them from the payments that it has received that day from the customers. This can be seen in listing 10.

4 Design of Manufacturer Agent Control Strategy

In relation to accepting or rejecting orders, when the manufacturer receives a new order from a customer, it is going to request the warehouse to calculate what would it cost to assemble and ship it. After calculating the total cost, if what the customer is going to pay for it provides to the manufacturer a minimum benefit margin of x% then it will accept it. In [1], a negotiation model for trading in a market is proposed in which customers select offers based on multiple criteria. Even if the system proposed in [1]

is much more complex than the scope of this project, the principle of selecting offers or orders based on multiple elements can still be applied. The way in which the total cost is calculated in this coursework takes into account the different prices from ordering components from different suppliers, how long will take for the new order to be assembled, if the new order can be assembled before any of the ones already pending without affecting them and the penalties for late deliveries. In addition to those, the criteria of a certain minimum benefit is also taking into account to discard orders that will provide low profits.

About how many components to order from each supplier, this decision is going to be made in a per order basis and it is going to depend in the results from calculating two different prices for the two different suppliers. This calculation is not only going to take into account the price differences between the components, it is also going to add to that price any penalties for late delivery based on when the order should be ready considering: when the components would arrived and the orders that are waiting to be assembled. The main problem of ordering components resides in whether to order more or just the ones that are strictly needed, trying to predict the demand of products in SCM is known as forecasting. In [6], is shown that forecasting plays a critical role in the success of a supply chain. The book [2] explores different methods to implement forecasting in supply chains, most of the methods proposed consider external factors like trend variations, experience of forecasters, historical data, etc. Those elements are outside the scope of this coursework, nevertheless, one of the methods called Naive Forecast does not requires any of those external factors as it assumes that the current demand is going to be equal to the one in the previous period. In this project, the customers decide in a completely random way what they are going to order, meaning that keeping track of any kind of data is not going to be useful. For this reason and in order to keep things a bit simpler, even if naive forecasting could be implemented, the warehouse is only going to order the components that it needs.

When it comes to which components place in the warehouse as opposed to using immediately, the orders that customers placed will be arranged by the warehouse in an ordered list in which the orders that need to be assemble first will be at the beginning and the orders that can wait will be at the end. Therefore, when the components are received from the supplier the warehouse will try to assemble as much orders as it can taking into account the order priority and the limit of fifty smartphones per day. After that, the components that have not been used will be stored. The book [2] also explores different methods for inventory management. In this project, in addition to what has been stated above, the approach to minimize the amount of elements stored in the warehouse to reduce costs is going to consist in the following: if an order is waiting for supplies from different suppliers and if those suppliers have different delivery times, the requests to buy supplies from the faster suppliers will be delayed with the purpose of making sure that all components arrive at the same time and that they are not stored in the stock more than what they need to.

Finally, in regarding to which orders assemble and ship, as stated before, the warehouse will keep track of all the orders in a list that will be ordered prioritizing some orders over others, whenever a new order comes, the warehouse will decide its position in the list. Consequently, the orders closer to the beginning of the list will be assembled and shipped first.

5 Experimental Results

For the evaluation of the agent control strategy, the profit will be calculated thirty times with different parameters to see how they affect the final result. The parameters that will be tested are: the per-day per-component cost of warehouse storage w, the number of customers c, the range of per-day penalties p and the benefit margin b. The reason for choosing these parameters is that they are the ones that affect directly the profit calculation meaning that, modifying them, could have a positive or negative effect on the profit. The increase of c should have a positive effect, the increase of w and p should have a negative effect and the increase of b might have a positive effect until it reaches a certain value. The default settings for the parameters are: $w = 5$, $c = 3$, $p = 1 - 50$ and $b = 20$.

The first parameter tested was w and the results were:

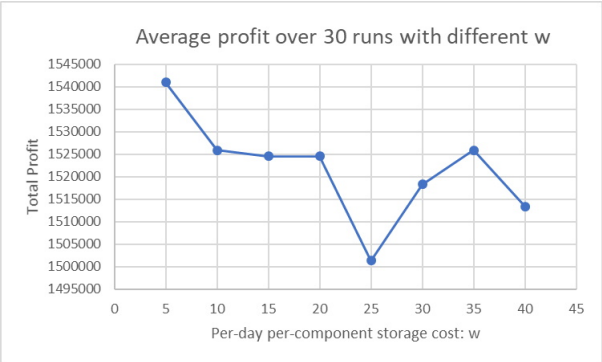

Figure 1: Total profit calculated with different per-day per-component storage cost.

From the results shown in figure 1, it can be seen that all the total profits are between 1500000 and 1545000, the total profit remains more or less stable no matter what w is, this means that the strategy success at minimizing the number of elements stored in the warehouse which results in w not having neither a positive nor negative effect.

The second parameter tested was c and the results were:

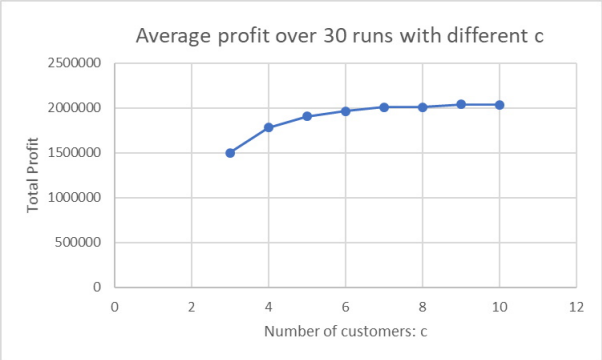

Figure 2: Total profit calculated with different number of customers.

It can be seen in figure 2 that increasing the number of customers also increases the total profit. However, the total profit ends up stabilizing, the reason behind this is probably that it comes to a point where the strategy cannot handle more orders, which means that it does not matter if there are more customers or not. This could be solved by allowing the warehouse to assemble more smartphones per day or increasing the number of warehouses.

The third parameter tested was p and the results were:

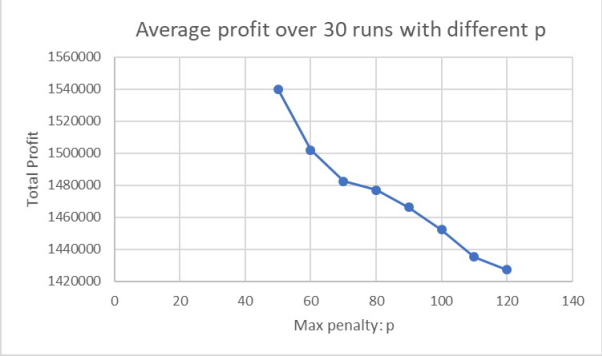

Figure 3: Total profit calculated with different penalty ranges (max penalties).

As shown in figure 3, increasing the maximum penalty that the customer can set to the order it requests is going to have a negative effect in the total profit.

The final parameter tested was b and the results were:

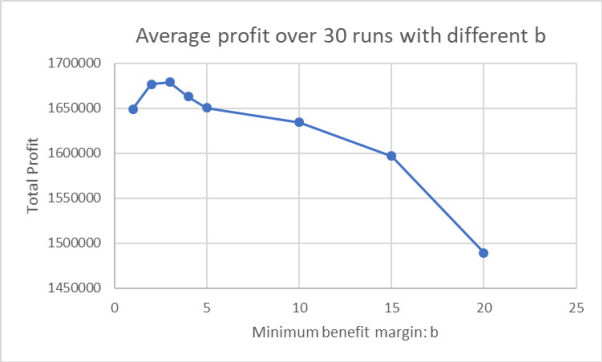

Figure 4: Total profit calculated with different minimum benefit margin.

As evidenced in figure 4, restricting the orders that the manufacturer accepts by increasing the minimum benefit margin has a small positive effect when the minimum margin is not too high, nevertheless, after a certain point, increasing the minimum benefit margin is going to have a negative effect in the total profit.

6 Conclusions

The supply chain management system represented in this implementation is a very simplistic version of what it would look in real life. Some of the elements that could be added to the system to make it handle a more realistic situation are the following:

- Introduce new customers at run time and allow them to buy more types of smartphones.
- Add a confidence paramater in the customer agents that would decrease when the manufacturer delivers orders late. This parameter would affect the amount of money the customer is willing to pay for an order or even make the customer refuse to buy from the manufacturer exiting the simulation.
- Try to simulate market patterns by making the customers have some probability of buying the same type of smartphone multiple times and having that probability to change occasionally.
- Allow the addition of new suppliers with different delivery times and components to sell at run time.
- Allow the suppliers to change the prices of the components they sell, to change the components they have available and to run out of their own components stock at run time.
- Allow the manufacturer to contract new warehouses, when needed, at run time so it can handle more orders.
- Introduce a stock limit in the warehouse and a renting cost for the warehouse itself.
- Introduce warehouses of different sizes that would have different stock sizes, renting costs and different maximum amount of smartphones to assemble per day.

In relation to the manufacturer strategy and taking into account a more realistic scenario with the elements proposed above, the first thing that could improve the strategy is the introduction of forecasting, this would mean that the manufacturer should try to identify smartphones that are bought more frequently so it can order components in advance. With the addition of forecasting, the strategy could also be improved with the introduction of inventory management methods. Finally, the last suggestion that could improve the strategy is allowing it to change at run time if it does not manage to acquire enough profit.

References

[1] A Study of Multi-Agent Based Supply Chain Modeling and Management. *iBusiness*, 02(04):333–341, 2010.

[2] Klaus-Dieter Gronwald. *SCM: Supply Chain Management*. 2017.

[3] Asoka Karunananda and LCM Perera. Using a multi-agent system for supply chain management. *International Journal of Design Nature and Ecodynamics*, 11(2), 2016.

[4] Keonsoo Lee, Wonil Kim, and Minkoo Kim. A review of supply chain management using multi-agent system. *International Journal of Computer Science Issues*, 7(5), 2004.

[5] Ksenija Mandic and Boris Delibašić. Application of multi-agent systems in supply chain management. *Management - Journal for theory and practice of management*, 17, 2012.

[6] Alan Smith and O. Offodile. Exploring forecasting and project management characteristics of supply chain management. *International Journal of Logistics Systems and Management - Int J Logist Syst Manag*, 3, 2007.

Appendices

A Ontology

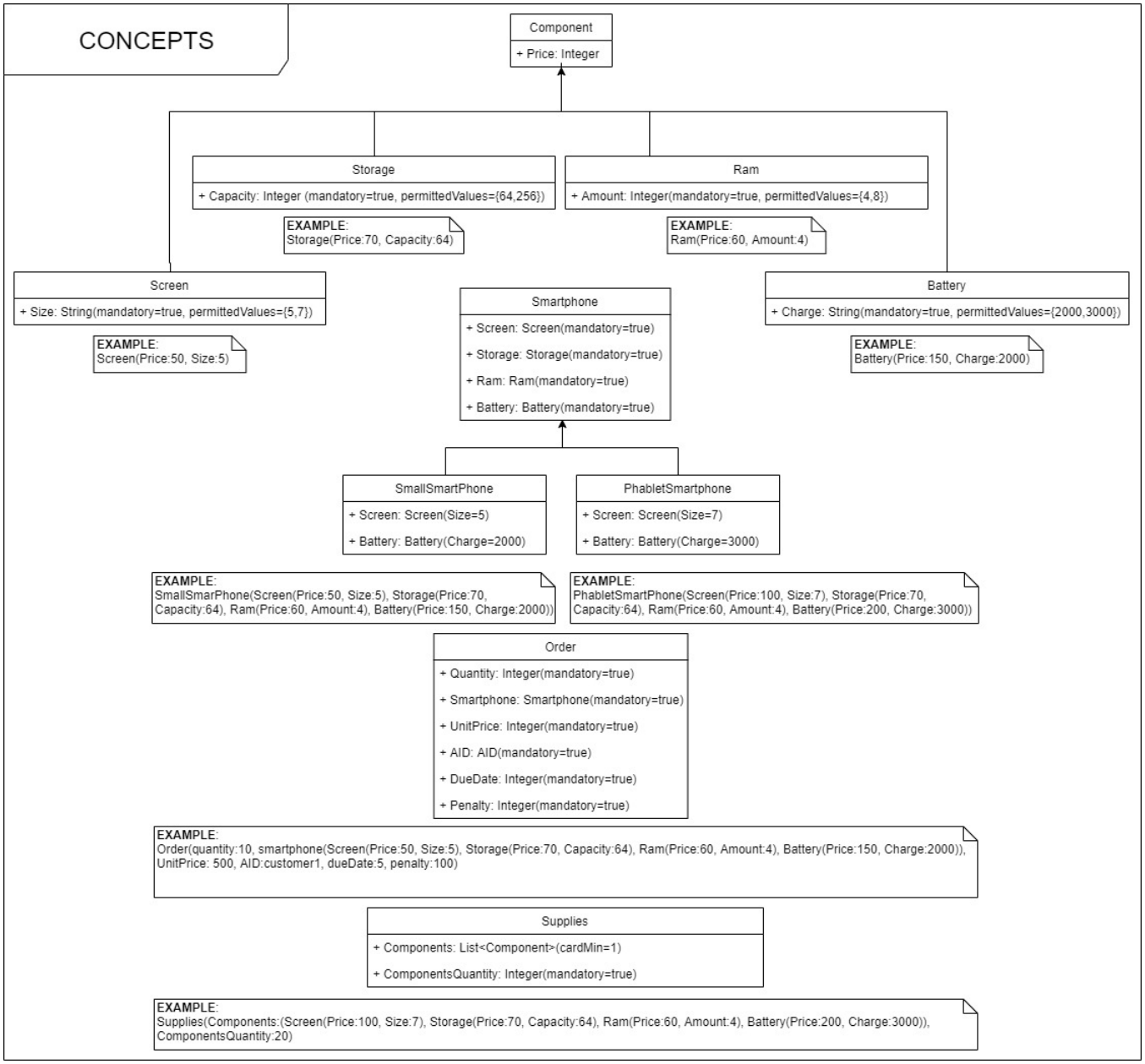

Figure 5: Ontology Concepts

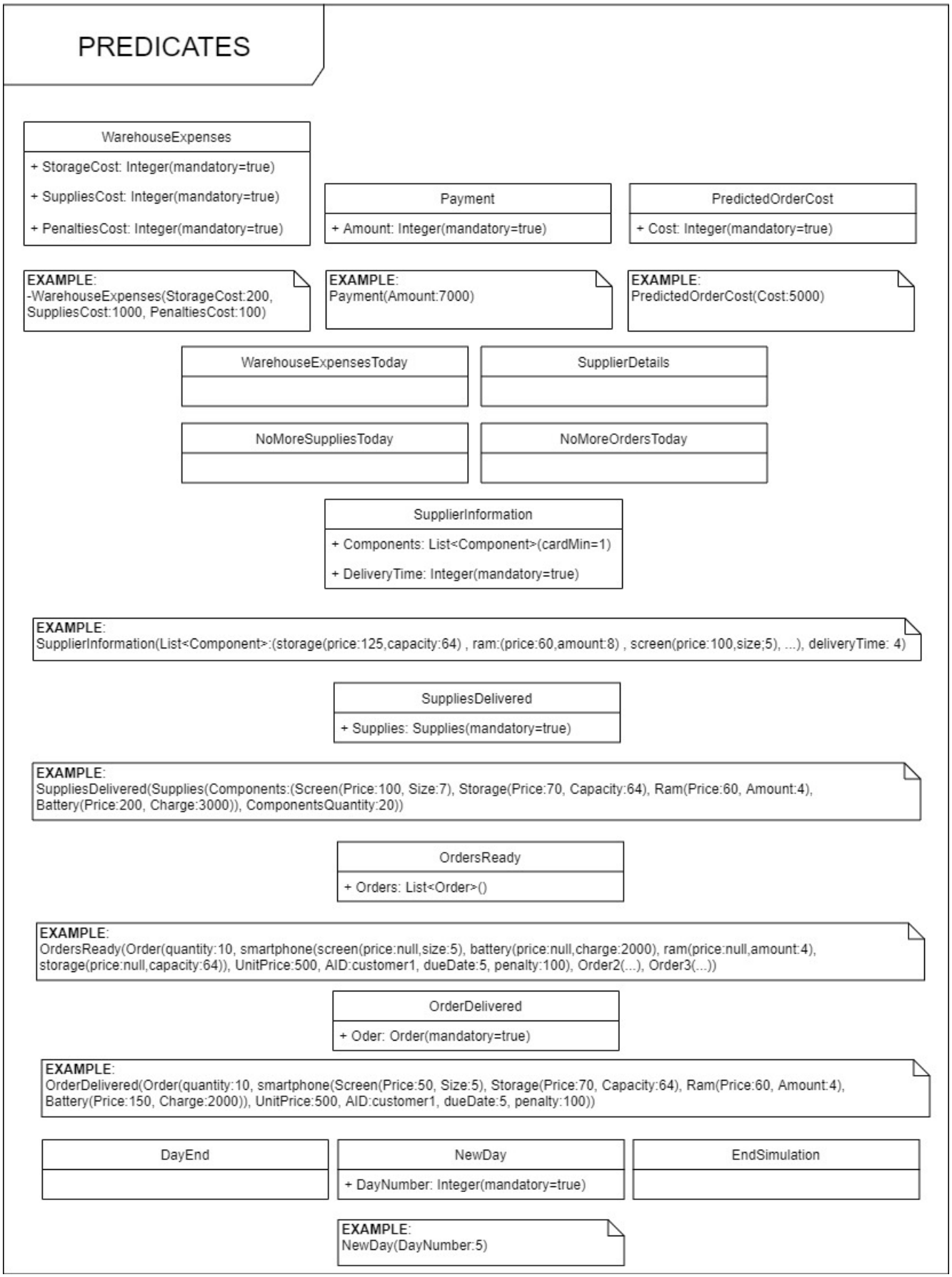

Figure 6: Ontology Predicates

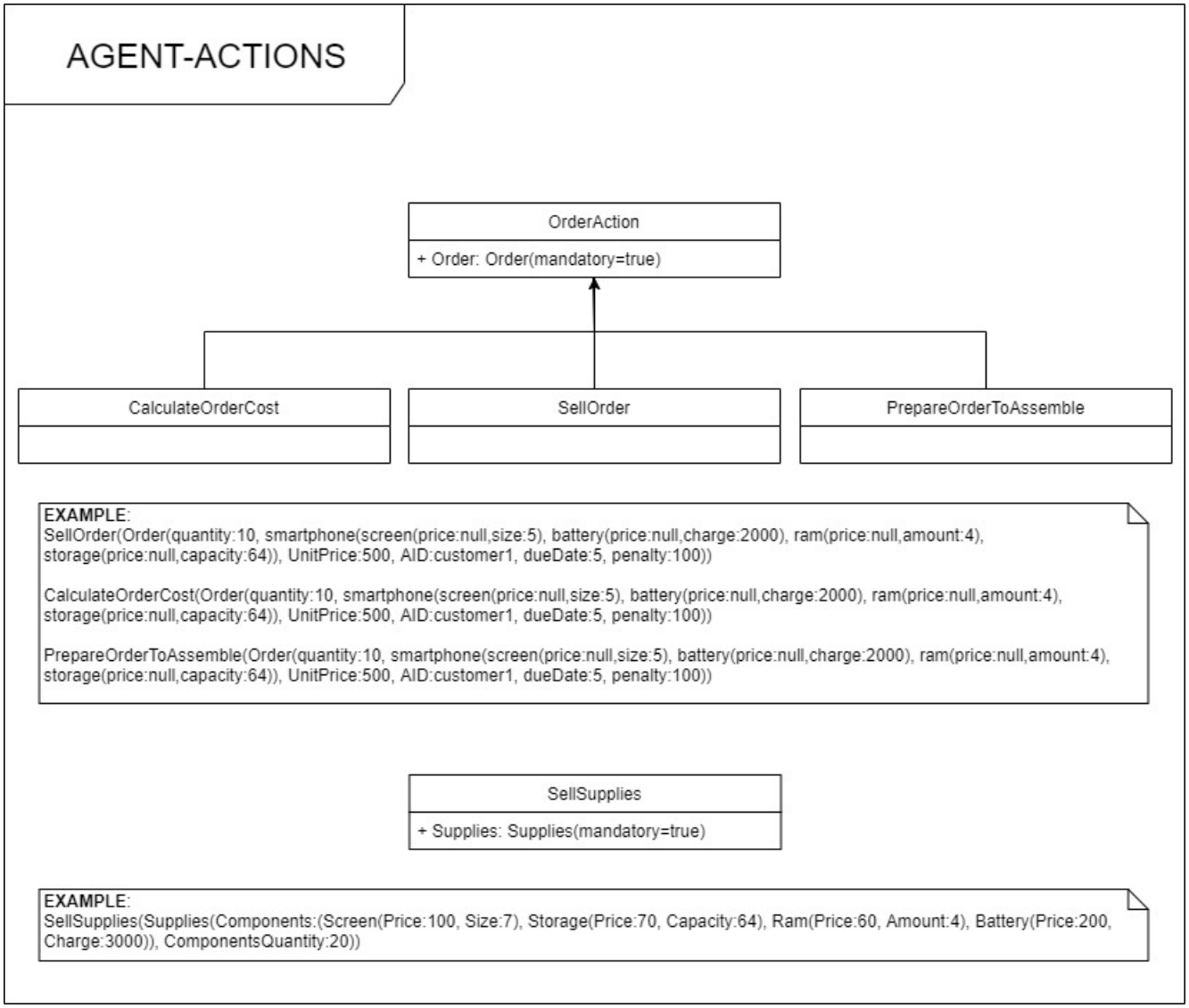

Figure 7: Ontology Agent-Actions

B Communication Protocols

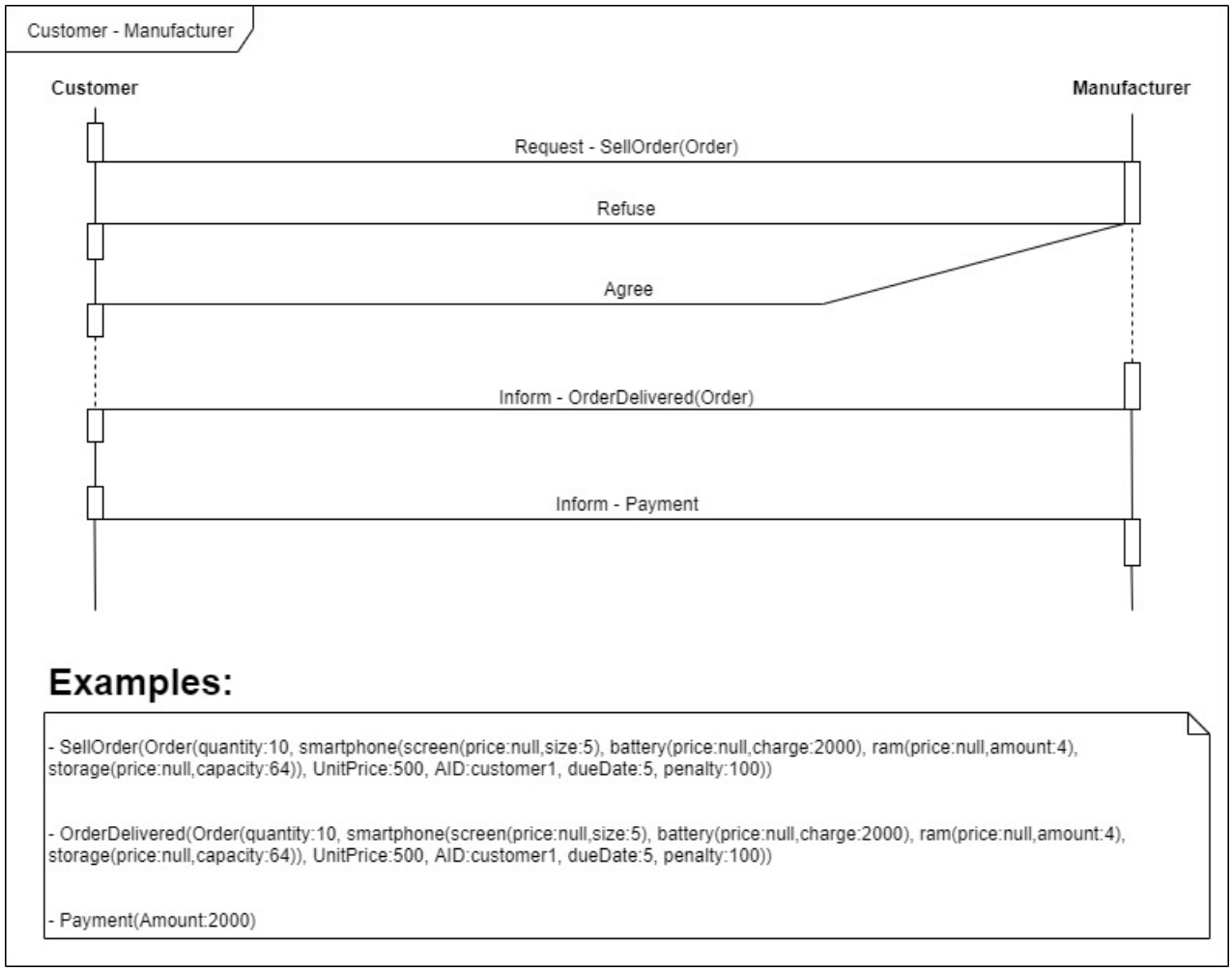

Figure 8: Customer-Manufacturer Communication

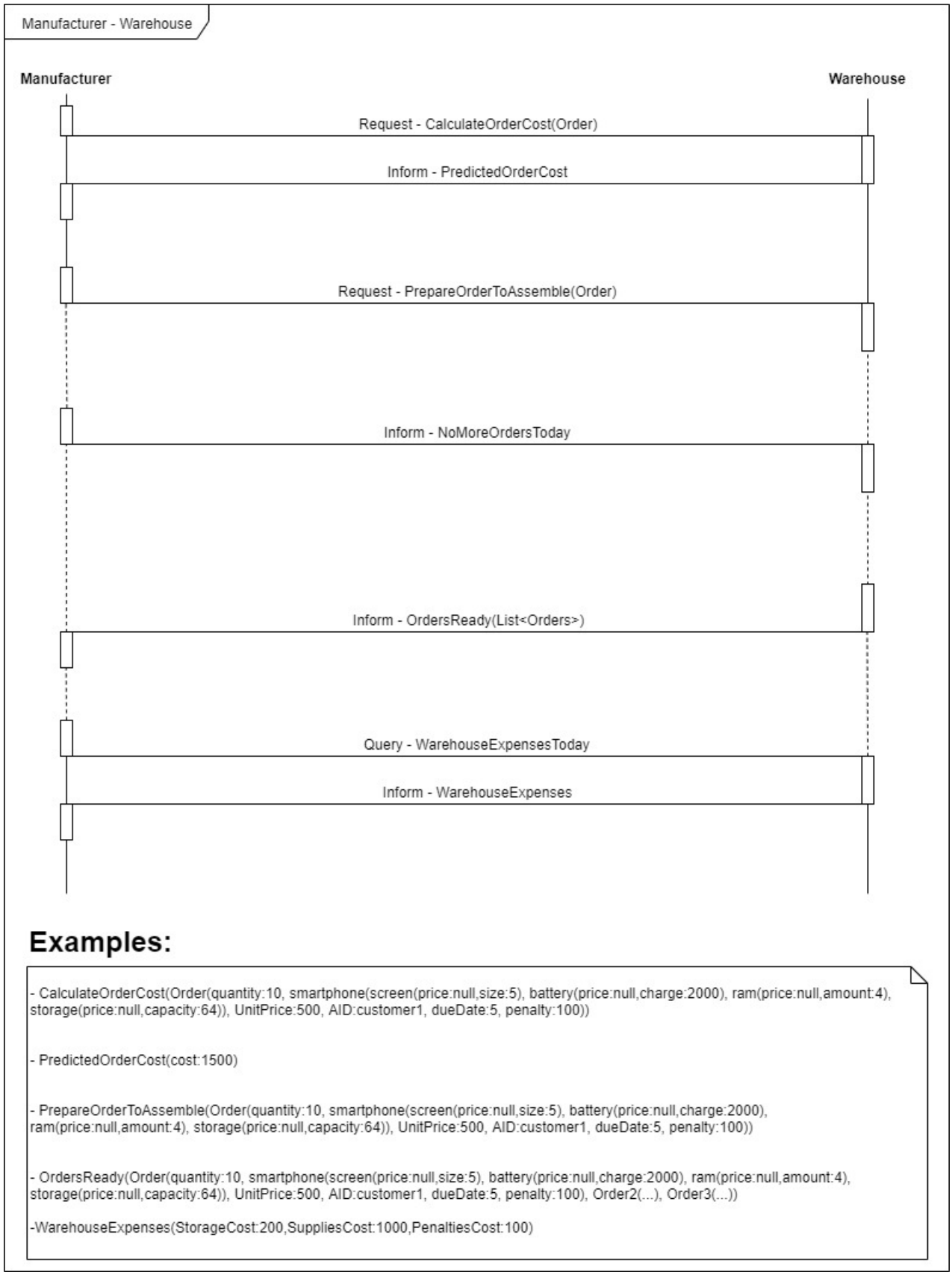

Figure 9: Manufacturer-Warehouse Communication

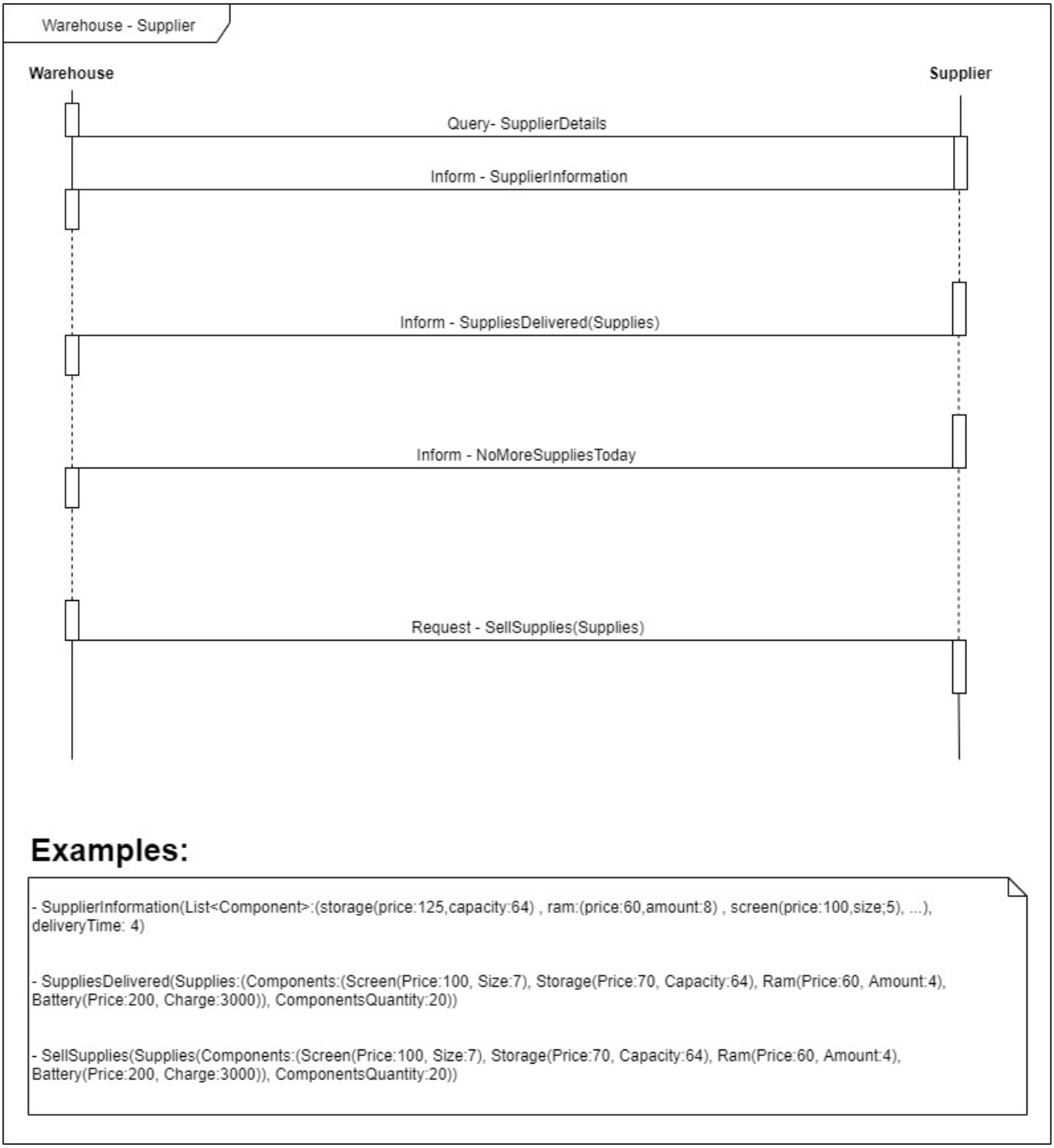

Figure 10: Warehouse-Supplier Communication

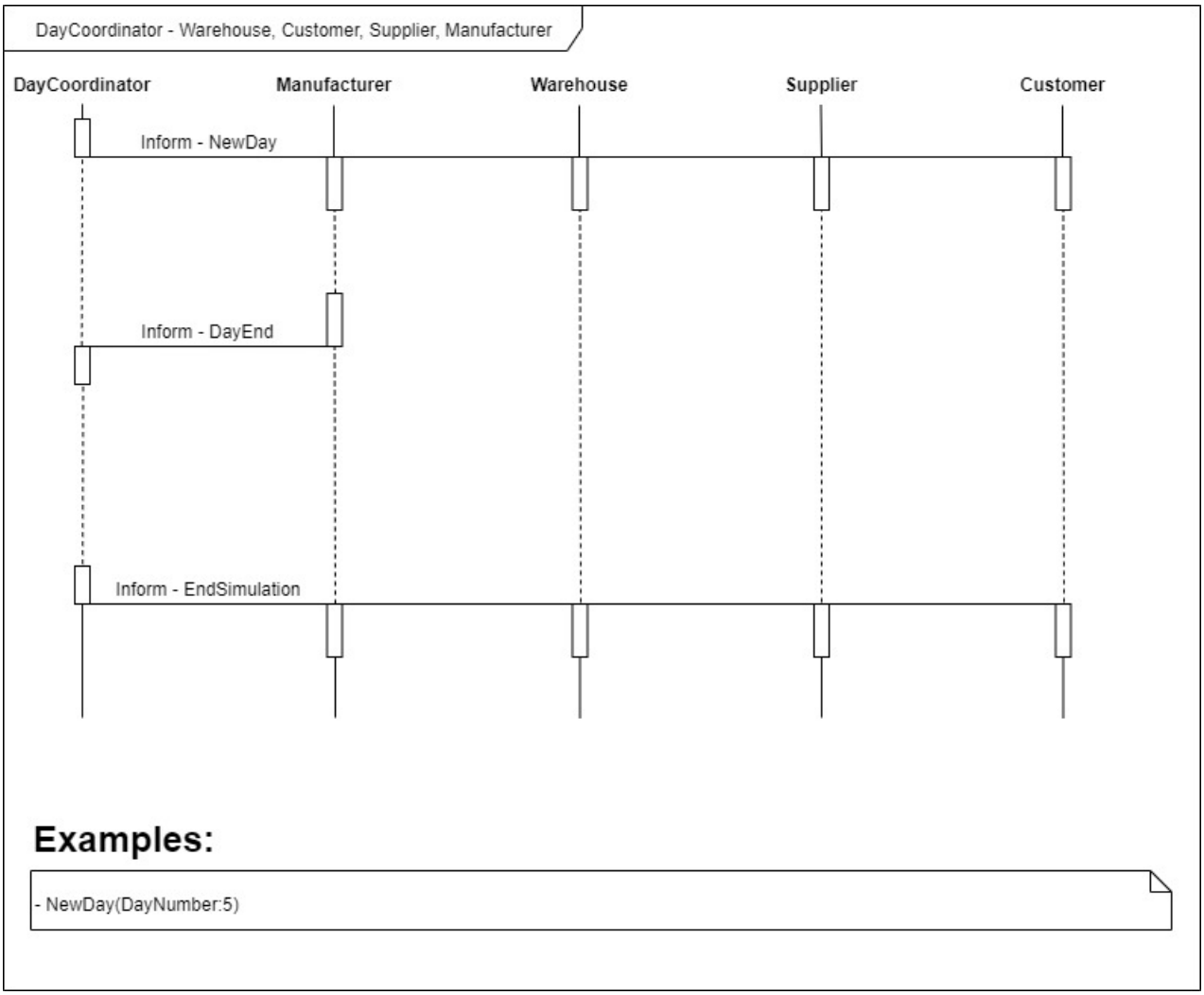

Figure 11: DayCoordinator-Warehouse,Customer,Manufacturer,Supplier Communication

C Source Code

Listing 1: Example of synchronizing/waiter behaviour

```
1     // Behaviour that waits for new day or end simulation calls
2     public class DayCoordinatorWaiterBehaviour extends CyclicBehaviour{
3         public void action() {
4             MessageTemplate mt = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.INFORM), ↩
      MessageTemplate.MatchSender(dayCoordinatorAID));
5             ACLMessage msg = myAgent.receive(mt);
6             if(msg != null) {
7                 try {
8                     // Convert string to java objects
9                     ContentElement ce = getContentManager().extractContent(msg);;
10                    // Every new day the warehouse is going to carry out a series of operations
11                    if(ce instanceof NewDay) {
12                        // Reset variables
13                        numbSmartphonesAssembledToday = 0;
14                        dailySuppliesPurchasedCost = 0;
15                        dailyPenaltiesCost = 0;
16                        dailyStorageCost = 0;
17                        // Add sequential behaviour
18                        SequentialBehaviour dailyActivity = new SequentialBehaviour();
19                        if(suppliers.isEmpty()) {
20                            dailyActivity.addSubBehaviour(new GetSupplierDetailsBehaviour());
21                        }
22                        dailyActivity.addSubBehaviour(new ProcessSuppliesDeliveredBehaviour());
23                        dailyActivity.addSubBehaviour(new ProcessPendingOrdersBehaviour());
24                        dailyActivity.addSubBehaviour(new UpdatePendingOrdersTimesAndCalculatePenaltiesBehaviour());
25                        dailyActivity.addSubBehaviour(new ProcessManufacturerOrderRequestsBehaviour());
26                        dailyActivity.addSubBehaviour(new RequestSuppliesToSuppliersBehaviour());
27                        dailyActivity.addSubBehaviour(new SendReadyOrdersBehaviour());
28                        dailyActivity.addSubBehaviour(new CalculateStorageCostBehaviour());
29                        dailyActivity.addSubBehaviour(new SendDailyWarehouseExpensesBehaviour());
30                        myAgent.addBehaviour(dailyActivity);
31                    } else {
32                        myAgent.doDelete();
33                    }
34
35                } catch (CodecException ce) {
36                    ce.printStackTrace();
37                } catch (OntologyException oe) {
38                    oe.printStackTrace();
39                }
40            } else {
41                block();
42            }
43        }
44    }
45
```

Listing 2: DayCoordinatorAgent SyncAgentsBehaviour

```
1       // Behaviour to synchronize agents
2       public class SyncAgentsBehaviour extends Behaviour{
3           private int step = 0;
4           private int currentDay = 1;
5
6           @Override
7           public void action() {
8               switch(step) {
9               case 0:
10                  // Create new day predicate
11                  NewDay newDay = new NewDay();
12                  newDay.setDayNumber(currentDay);
13                  // Send new day message to each agent
14                  ACLMessage msgNewDay = new ACLMessage(ACLMessage.INFORM);
15                  msgNewDay.setLanguage(codec.getName());
16                  msgNewDay.setOntology(ontology.getName());
17                  msgNewDay.addReceiver(manufacturerAID);
18                  msgNewDay.addReceiver(warehouseAID);
19                  for(AID aid : customersAID) {
20                      msgNewDay.addReceiver(aid);
21                  }
22                  for(AID aid : suppliersAID) {
23                      msgNewDay.addReceiver(aid);
24                  }
25                  try {
26                      // Convert from java objects to string
27                      getContentManager().fillContent(msgNewDay, newDay);
28                      myAgent.send(msgNewDay);
29                  } catch (CodecException ce) {
30                       ce.printStackTrace();
31                  } catch (OntologyException oe) {
32                   oe.printStackTrace();
33                  }
34                  step++;
35                  currentDay++;
36                  break;
37              case 1:
38                  // Wait for message from manufacturer to arrive
39                  MessageTemplate mt = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.INFORM), ↩
    MessageTemplate.MatchSender(manufacturerAID));
40                  ACLMessage msg = myAgent.receive(mt);
41                  if(msg != null) {
42                      try {
43                          ContentElement ce = getContentManager().extractContent(msg);
44                          if(ce instanceof DayEnd) {
45                              System.out.println("End of day: " + currentDay);
46                              step++;
47                          }
48                      } catch (CodecException ce) {
49                          ce.printStackTrace();
50                      } catch (OntologyException oe) {
51                          oe.printStackTrace();
52                      }
53                  } else {
54                      block();
55                  }
56                  break;
57              }
58          }
59
60          @Override
61          public boolean done() {
62              return step == 2;
63          }
64
65          @Override
66          public void reset() {
67              super.reset();
68              step = 0;
69          }
70
71          @Override
72          public int onEnd() {
73              if(currentDay == NUM_DAYS) {
74                  // Create end simulation predicate
75                  EndSimulation endSimulation = new EndSimulation();
76                  // Send end simulation message to each agent
77                  ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
78                  msg.setLanguage(codec.getName());
79                  msg.setOntology(ontology.getName());
80                  msg.addReceiver(manufacturerAID);
```

```
81                  msg.addReceiver(warehouseAID);
82                  for(AID aid : customersAID) {
83                      msg.addReceiver(aid);
84                  }
85                  for(AID aid : suppliersAID) {
86                      msg.addReceiver(aid);
87                  }
88                  try {
89                      // Convert from java objects to string
90                      getContentManager().fillContent(msg, endSimulation);
91                      myAgent.send(msg);
92                  } catch (CodecException ce) {
93                       ce.printStackTrace();
94                  } catch (OntologyException oe) {
95                   oe.printStackTrace();
96                  }
97                  // Delete this agent
98                  myAgent.doDelete();
99              } else {
100                 reset();
101                 myAgent.addBehaviour(this);
102             }
103             return 0;
104         }
105     }
106
```

Listing 3: CustomerAgent RequestOrderBehaviour

```
1   // Behaviour to request an order to the manufacturer
2   public class RequestOrderBehaviour extends OneShotBehaviour{
3       public void action() {
4           // Create new order
5           Order order = createOrder(myAgent.getAID());
6           // Create agent action
7           SellOrder sellOrder = new SellOrder();
8           sellOrder.setOrder(order);
9           // Create wrapper
10          Action request = new Action();
11          request.setAction(sellOrder);
12          request.setActor(manufacturerAID);
13          // Send request to manufacturer
14          ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
15          msg.addReceiver(manufacturerAID);
16          msg.setLanguage(codec.getName());
17          msg.setOntology(ontology.getName());
18          try {
19              // Convert java object to string
20              getContentManager().fillContent(msg, request);
21              myAgent.send(msg);
22          } catch (CodecException ce) {
23               ce.printStackTrace();
24          } catch (OntologyException oe) {
25               oe.printStackTrace();
26          }
27      }
28
29  }
30
```

Listing 4: CustomerAgent ReceiveOrderBehaviour

```
1   // Behaviour that is going to handle orders being received
2   public class ReceiveOrderBehaviour extends CyclicBehaviour{
3       public void action() {
4           MessageTemplate mt = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.INFORM), ↩
    MessageTemplate.MatchSender(manufacturerAID));
5           ACLMessage msg = myAgent.receive(mt);
6           if(msg != null) {
7               // Receive order
8               try {
9                   // Convert string to java objects
10                  ContentElement ce = getContentManager().extractContent(msg);
11                  if(ce instanceof OrderDelivered) {
12                      OrderDelivered orderDelivered = (OrderDelivered) ce;
13                      // Create predicate
14                      Payment payment = new Payment();
```

```
                    payment.setAmount(orderDelivered.getOrder().getQuantity() * orderDelivered.getOrder().getUnitPrice());
                    // Send payment
                    ACLMessage paymentMsg = new ACLMessage(ACLMessage.INFORM);
                    paymentMsg.addReceiver(msg.getSender());
                    paymentMsg.setLanguage(codec.getName());
                    paymentMsg.setOntology(ontology.getName());
                    try {
                        // Convert java objects to strings
                        getContentManager().fillContent(paymentMsg, payment);
                        myAgent.send(paymentMsg);
                    } catch (CodecException codece) {
                        codece.printStackTrace();
                    } catch (OntologyException oe) {
                        oe.printStackTrace();
                    }
                }

            } catch (CodecException ce) {
                ce.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
        } else {
            block();
        }
    }
}
```

Listing 5: SupplierAgent ProvideDetailsBehaviour

```
// Behaviour that is going to provide to the sender with this agent details, this will only happen once
public class ProvideDetailsBehaviour extends Behaviour{
    private boolean detailsSent = false;
    public void action() {
        MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF);
        ACLMessage msg = myAgent.receive(mt);
        if(msg != null) {
            try {
                // Convert string to java objects
                ContentElement ce =  getContentManager().extractContent(msg);
                if(ce instanceof SupplierDetails) {
                    // Send a message to the manufacturer with the supplier information
                    ACLMessage response = new ACLMessage(ACLMessage.INFORM);
                    response.addReceiver(warehouseAID);
                    response.setLanguage(codec.getName());
                    response.setOntology(ontology.getName());
                    try {
                        // Convert java objects to string
                        getContentManager().fillContent(response, supplierInformation);
                        myAgent.send(response);
                    } catch (CodecException codece) {
                        codece.printStackTrace();
                    } catch (OntologyException oe) {
                        oe.printStackTrace();
                    }
                    detailsSent = true;
                }
            } catch (CodecException ce) {
                ce.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
        } else {
            block();
        }
    }

    public boolean done() {
        return detailsSent;
    }
}
```

Listing 6: SupplierAgent ProcessSuppliesRequestsBehaviour

```
// Behaviour that is going to process requests to sell supplies
public class ProcessSuppliesRequestsBehaviour extends CyclicBehaviour{
    public void action() {
        MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
        ACLMessage msg = myAgent.receive(mt);
        if(msg != null) {
            try {
                // Convert strings to java objects
                ContentElement ce = getContentManager().extractContent(msg);
                if(ce instanceof Action) {
                    Concept action = ((Action)ce).getAction();
                    if(action instanceof SellSupplies) {
                        SellSupplies sellSupplies = (SellSupplies) action;
                        // Add the supplies to the list of pending deliveries
                        SuppliesToDeliver suppliesToDeliver = new SuppliesToDeliver();
                        suppliesToDeliver.setDaysLeftToDeliver(supplierInformation.getDeliveryTime());
                        suppliesToDeliver.setSupplies(sellSupplies.getSupplies());
                        pendingDeliveries.add(suppliesToDeliver);
                    }
                }
            } catch (CodecException codece) {
                codece.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
        }
    }
}

```

Listing 7: SupplierAgent SendSuppliesBehaviour

```
// Behaviour that is going to send supplies if they are ready at the beginning of each day
public class SendSuppliesBehaviour extends OneShotBehaviour{
    public void action() {
        ArrayList<SuppliesToDeliver> deliveriesToRemove = new ArrayList<SuppliesToDeliver>();
        for(SuppliesToDeliver delivery : pendingDeliveries) {
            int days = delivery.getDaysLeftToDeliver();
            days--;
            if(days == 0) {
                // Create predicate
                SuppliesDelivered suppliesDelivered = new SuppliesDelivered();
                suppliesDelivered.setSupplies(delivery.getSupplies());
                // Send supplies to warehouse
                ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
                msg.addReceiver(warehouseAID);
                msg.setLanguage(codec.getName());
                msg.setOntology(ontology.getName());
                try {
                    getContentManager().fillContent(msg, suppliesDelivered);
                    myAgent.send(msg);
                } catch (CodecException codece) {
                    codece.printStackTrace();
                } catch (OntologyException oe) {
                    oe.printStackTrace();
                }
                // Add delivery to the list of deliveries to remove
                deliveriesToRemove.add(delivery);
            } else {
                // Update days left to deliver
                delivery.setDaysLeftToDeliver(days);
            }
        }
        // Remove deliveries from list of pending deliveries
        pendingDeliveries.removeAll(deliveriesToRemove);
        // Create Predicate
        NoMoreSuppliesToday noMoreSuppliesToday = new NoMoreSuppliesToday();
        // Inform warehouse that there are no more supplies to deliver
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.addReceiver(warehouseAID);
        msg.setLanguage(codec.getName());
        msg.setOntology(ontology.getName());
        try {
            // Transform java objects to strings
            getContentManager().fillContent(msg, noMoreSuppliesToday);
            myAgent.send(msg);
```

```
45              } catch (CodecException codece) {
46                  codece.printStackTrace();
47              } catch (OntologyException oe) {
48                  oe.printStackTrace();
49              }
50          }
51      }
52
```

Listing 8: ManufacturerAgent ProcessOrderBehaviour

```
1       // Behaviour that is going to agree or refuse to process an order from a customer
2       private class ProcessOrderBehaviour extends Behaviour{
3           int step = 0;
4           int ordersReceived = 0;
5           Order currentOrder;
6
7           public void action() {
8               switch(step) {
9               // Receive a sell order request message from a customer
10              case 0:
11                  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
12                  ACLMessage msg = myAgent.receive(mt);
13                  if(msg != null) {
14                      try {
15                          ContentElement ce = getContentManager().extractContent(msg);
16                          if(ce instanceof Action) {
17                              Concept action = ((Action)ce).getAction();
18                              if(action instanceof SellOrder) {
19                                  SellOrder sellOrder = (SellOrder) action;
20                                  // Create agent action
21                                  CalculateOrderCost calculateOrderCost = new CalculateOrderCost();
22                                  calculateOrderCost.setOrder(sellOrder.getOrder());
23                                  // Create wrapper
24                                  Action request = new Action();
25                                  request.setAction(calculateOrderCost);
26                                  request.setActor(warehouseAID);
27                                  // Request warehouse to calculate order cost
28                                  ACLMessage costRequestMsg = new ACLMessage(ACLMessage.REQUEST);
29                                  costRequestMsg.addReceiver(warehouseAID);
30                                  costRequestMsg.setLanguage(codec.getName());
31                                  costRequestMsg.setOntology(ontology.getName());
32                                  try {
33                                      // Convert java objects to strings
34                                      getContentManager().fillContent(costRequestMsg, request);
35                                      myAgent.send(costRequestMsg);
36                                  } catch (CodecException codece) {
37                                      codece.printStackTrace();
38                                  } catch (OntologyException oe) {
39                                      oe.printStackTrace();
40                                  }
41                                  currentOrder = sellOrder.getOrder();
42                                  step++;
43                                  ordersReceived++;
44                              }
45                          }
46                      } catch (CodecException ce) {
47                          ce.printStackTrace();
48                      } catch (OntologyException oe) {
49                          oe.printStackTrace();
50                      }
51                  }
52                  break;
53              // Receive costs message from the warehouse
54              case 1:
55                  MessageTemplate mt1 = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
56                  ACLMessage msg1 = myAgent.receive(mt1);
57                  if( msg1 != null) {
58                      try {
59                          ContentElement ce = getContentManager().extractContent(msg1);
60                          if(ce instanceof PredictedOrderCost) {
61                              PredictedOrderCost costs = (PredictedOrderCost) ce;
62                              // Extact benefit from the costs and the sell price
63                              int sellPrice = currentOrder.getUnitPrice() * currentOrder.getQuantity();
64                              int benefit = sellPrice - costs.getCost();
65                              // Decide whether to accept the order or not based on the minimum benefit margin
66                              float benefitMargin = ((float)benefit / (float)sellPrice) * 100.0f;
67                              if(benefitMargin >= minimumBenefitMargin) {
68                                  // Create action
69                                  PrepareOrderToAssemble orderToAssemble = new PrepareOrderToAssemble();
70                                  orderToAssemble.setOrder(currentOrder);
```

```
                        // Create wraper
                        Action request = new Action();
                        request.setAction(orderToAssemble);
                        request.setActor(warehouseAID);
                        // Request warehouse to prepare the order to assemble
                        ACLMessage prepareOrderRequestMsg = new ACLMessage(ACLMessage.REQUEST);
                        prepareOrderRequestMsg.addReceiver(warehouseAID);
                        prepareOrderRequestMsg.setLanguage(codec.getName());
                        prepareOrderRequestMsg.setOntology(ontology.getName());
                        try {
                            // Transform java objects to strings
                            getContentManager().fillContent(prepareOrderRequestMsg, request);
                            myAgent.send(prepareOrderRequestMsg);
                        } catch (CodecException codece) {
                            codece.printStackTrace();
                        } catch (OntologyException oe) {
                            oe.printStackTrace();
                        }
                    }
                    // Check if there are more orders to receive from customers
                    if(ordersReceived == customersAID.length) {
                        step++;
                    } else {
                        step = 0;
                    }
                }
            } catch (CodecException ce) {
                ce.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
        }
        break;
    // Send notification to warehouse about no more orders for the day
    case 2:
        // Create predicate
        NoMoreOrdersToday  noMoreOrdersToday = new NoMoreOrdersToday();
        // Send message to warehouse
        ACLMessage noMoreOrdersInformMsg = new ACLMessage(ACLMessage.INFORM);
        noMoreOrdersInformMsg.addReceiver(warehouseAID);
        noMoreOrdersInformMsg.setLanguage(codec.getName());
        noMoreOrdersInformMsg.setOntology(ontology.getName());
        try {
            // Transform java objects to string
            getContentManager().fillContent(noMoreOrdersInformMsg, noMoreOrdersToday);
            myAgent.send(noMoreOrdersInformMsg);
        } catch (CodecException ce) {
            ce.printStackTrace();
        } catch (OntologyException oe) {
            oe.printStackTrace();
        }
        step++;
        break;
    }
}

public boolean done() {
    return step == 3;
}
}
```

Listing 9: ManufacturerAgent ProcessOrdersReadyBehaviour

```
// Behaviour that is going to receive the orders ready from the warehouse, send them to the customers and receive the corresponding payment
private class ProcessOrdersReadyBehaviour extends Behaviour{
    private int step = 0;
    private int numbOrdersReadyDelivered = 0;
    private int numbPaymentsReceived = 0;

    public void action() {
        switch(step) {
        // Receive orders ready from the warehouse
        case 0:
            MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
            ACLMessage msg = myAgent.receive(mt);
            if(msg != null) {
                try {
                    // Transform string to java objects
                    ContentElement ce = getContentManager().extractContent(msg);
```

```
17                      if(ce instanceof OrdersReady) {
18                          OrdersReady ordersReady = (OrdersReady) ce;
19                          // If there are any orders ready send them to the according customer
20                          if(ordersReady.getOrders() == null || ordersReady.getOrders().isEmpty()) {
21                              step = 2;
22                          }
23                          else
24                          {
25                              for(Order orderReady : ordersReady.getOrders()) {
26                                  // Create predicate
27                                  OrderDelivered orderDelivered = new OrderDelivered();
28                                  orderDelivered.setOrder(orderReady);
29                                  // Send order ready to customer
30                                  ACLMessage orderDeliveredMsg = new ACLMessage(ACLMessage.INFORM);
31                                  orderDeliveredMsg.addReceiver(orderReady.getAID());
32                                  orderDeliveredMsg.setLanguage(codec.getName());
33                                  orderDeliveredMsg.setOntology(ontology.getName());
34                                  try {
35                                      // Transform java objects to string
36                                      getContentManager().fillContent(orderDeliveredMsg, orderDelivered);
37                                      myAgent.send(orderDeliveredMsg);
38                                  } catch (CodecException codece) {
39                                      codece.printStackTrace();
40                                  } catch (OntologyException oe) {
41                                      oe.printStackTrace();
42                                  }
43                              }
44                              numbOrdersReadyDelivered = ordersReady.getOrders().size();
45                              step = 1;
46                          }
47                      } else {
48                          myAgent.postMessage(msg);
49                      }
50                  } catch (CodecException ce) {
51                      ce.printStackTrace();
52                  } catch (OntologyException oe) {
53                      oe.printStackTrace();
54                  }
55              } else {
56                  block();
57              }
58              break;
59          // Receive payment from customers
60          case 1:
61              MessageTemplate mt1 = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
62              ACLMessage msg1 = myAgent.receive(mt1);
63              if(msg1 != null) {
64                  try {
65                      // Transform strings to java objects
66                      ContentElement ce = getContentManager().extractContent(msg1);
67                      if(ce instanceof Payment) {
68                          // Add payment to daily payments
69                          Payment payment = (Payment) ce;
70                          dailyPayments += payment.getAmount();
71                          numbPaymentsReceived++;
72                          // When all payments are received increase step
73                          if(numbOrdersReadyDelivered == numbPaymentsReceived) {
74                              step++;
75                          }
76                      }
77                  }  catch (CodecException ce) {
78                      ce.printStackTrace();
79                  } catch (OntologyException oe) {
80                      oe.printStackTrace();
81                  }
82              } else {
83                  block();
84              }
85              break;
86          }
87      }
88
89      public boolean done() {
90          return step == 2;
91      }
92  }
93
```

Listing 10: ManufacturerAgent CalculateDailyProfitBehaviour

```
1   // Behaviour that is going to finish the calculation of the daily profit and add it to the total profit
2   private class CalculateDailyProfitBehaviour extends Behaviour{
3       private int step = 0;
4
5       public void action() {
6           switch(step) {
7           // Query the warehouse about all the costs of the day
8           case 0:
9               // Create predicate
10              WarehouseExpensesToday warehouseExpensesToday = new WarehouseExpensesToday();
11              // Send message
12              ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
13              msg.addReceiver(warehouseAID);
14              msg.setLanguage(codec.getName());
15              msg.setOntology(ontology.getName());
16              try {
17                  // Transform java objects to strings
18                  getContentManager().fillContent(msg, warehouseExpensesToday);
19                  myAgent.send(msg);
20              } catch (CodecException ce) {
21                  ce.printStackTrace();
22              } catch (OntologyException oe) {
23                  oe.printStackTrace();
24              }
25              step++;
26              break;
27          // Receive warehouse costs message from warehouse
28          case 1:
29              MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
30              ACLMessage msg1 = myAgent.receive(mt);
31              if(msg1 != null) {
32                  try {
33                      // Transform strings to java objects
34                      ContentElement ce = getContentManager().extractContent(msg1);
35                      if(ce instanceof WarehouseExpenses) {
36                          // Get storage cost for the day
37                          WarehouseExpenses expenses = (WarehouseExpenses) ce;
38                          dailyPurchasesCost = expenses.getSuppliesCost();
39                          dailyPenaltiesCost = expenses.getPenaltiesCost();
40                          dailyWarehouseStorageCost = expenses.getStorageCost();
41                          step++;
42                      }
43                  } catch (CodecException codece) {
44                      codece.printStackTrace();
45                  } catch (OntologyException oe) {
46                      oe.printStackTrace();
47                  }
48              }
49              break;
50          // Calculate daily profit
51          case 2:
52              dailyProfit = dailyPayments - dailyPenaltiesCost - dailyWarehouseStorageCost - dailyPurchasesCost;
53              totalProfit += dailyProfit;
54              System.out.println("Daily profit of: " + dailyProfit);
55              System.out.println("Total profit of: " + totalProfit);
56              step++;
57              break;
58          }
59      }
60
61      public boolean done() {
62          return step == 3;
63      }
64  }
65
```

Listing 11: ManufacturerAgent EndDayBehaviour

```
1   // Behaviour that is going to call the day off
2   private class EndDayBehaviour extends OneShotBehaviour{
3       public void action() {
4           // Reset daily variables
5           dailyProfit = 0;
6           dailyPurchasesCost = 0;
7           dailyPenaltiesCost = 0;
8           dailyWarehouseStorageCost = 0;
9           dailyPayments = 0;
```

```
10              // Create predicate
11              DayEnd dayEnd = new DayEnd();
12              // Send day end message to day coordinator
13              ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
14              msg.addReceiver(dayCoordinatorAID);
15              msg.setLanguage(codec.getName());
16              msg.setOntology(ontology.getName());
17              try {
18                  // Transform java objects to strings
19                  getContentManager().fillContent(msg, dayEnd);
20                  myAgent.send(msg);
21              } catch (CodecException codece) {
22                  codece.printStackTrace();
23              } catch (OntologyException oe) {
24                  oe.printStackTrace();
25              }
26          }
27      }
28
```

Listing 12: Warehouse GetSupplierDetailsBehaviour

```
1       // Behaviour that is going to ask the suppliers for their components, prices and delivery times
2       private class GetSupplierDetailsBehaviour extends Behaviour{
3           private int step = 0;
4           int numResponsesReceived = 0;
5
6           public void action() {
7               switch(step) {
8               // Send query message to suppliers
9               case 0:
10                  // Create predicate
11                  SupplierDetails supplierDetails = new SupplierDetails();
12                  // Send message
13                  ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
14                  for(int i = 0; i < suppliersAID.length; i++) {
15                      msg.addReceiver(suppliersAID[i]);
16                  }
17                  msg.setLanguage(codec.getName());
18                  msg.setOntology(ontology.getName());
19                  try {
20                      // Transform java objects to strings
21                      getContentManager().fillContent(msg, supplierDetails);
22                      myAgent.send(msg);
23                  } catch (CodecException ce) {
24                      ce.printStackTrace();
25                  } catch (OntologyException oe) {
26                      oe.printStackTrace();
27                  }
28                  step++;
29                  break;
30              // Receive suppliers response
31              case 1:
32                  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
33                  ACLMessage msg1 = myAgent.receive(mt);
34                  if(msg1 != null) {
35                      try {
36                          // Transform strings to java objects
37                          ContentElement ce = getContentManager().extractContent(msg1);
38                          if(ce instanceof SupplierInformation) {
39                              SupplierInformation supplierInformation = (SupplierInformation) ce;
40                              // Store the supplier information internally
41                              suppliers.add(new Supplier(supplierInformation.getComponents(), supplierInformation.getDeliveryTime(), msg1.↩
    getSender()));
42                              numResponsesReceived++;
43                              if(numResponsesReceived == suppliersAID.length) {
44                                  step++;
45                              }
46                          } else {
47                              myAgent.postMessage(msg1);
48                          }
49                      } catch (CodecException ce) {
50                          ce.printStackTrace();
51                      } catch (OntologyException oe) {
52                          oe.printStackTrace();
53                      }
54                  } else {
55                      block();
56                  }
57                  break;
58              }
```

```
59
60          }
61
62          public boolean done() {
63              return step == 2;
64          }
65      }
66
```

Listing 13: Warehouse ProcessSuppliesDeliveredBehaviour

```
1       // Behaviour that is going to process the supplies received by the supplier
2       private class ProcessSuppliesDeliveredBehaviour extends Behaviour{
3           private int numbSuppliersDone = 0;
4
5           public void action() {
6               MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
7               ACLMessage msg = myAgent.receive(mt);
8               if(msg != null) {
9                   try {
10                      // Transform strings to java objects
11                      ContentElement ce = getContentManager().extractContent(msg);
12                      if(ce instanceof SuppliesDelivered) {
13                          // Store supplies internally
14                          SuppliesDelivered suppliesDelivered = (SuppliesDelivered) ce;
15                          int quantityPerComponent = suppliesDelivered.getSupplies().getComponentsQuantity();
16                          for(Component component : suppliesDelivered.getSupplies().getComponents()) {
17                              // Check if the same component with the same price is already in stock map
18                              if(stock.containsKey(component)) {
19                                  // Increase the quantity
20                                  int quantity = quantityPerComponent + stock.get(component);
21                                  stock.replace(component, quantity);
22                              } else {
23                                  // Add the mapElement to the stock
24                                  stock.put(component, quantityPerComponent);
25                              }
26                          }
27                      } else if (ce instanceof NoMoreSuppliesToday) {
28                          // Each supplier is going to send the NoMoreseSuppliesToday predicate after sending all the supplies
29                          numbSuppliersDone++;
30                      } else {
31                          myAgent.postMessage(msg);
32                      }
33                  } catch (CodecException ce) {
34                      ce.printStackTrace();
35                  } catch (OntologyException oe) {
36                      oe.printStackTrace();
37                  }
38              } else {
39                  block();
40              }
41
42          }
43
44          public boolean done() {
45              return numbSuppliersDone == suppliersAID.length;
46          }
47      }
48
```

Listing 14: Warehouse ProcessManufacturerOrderRequestsBehaviour

```
1       // Behaviour that is going to calculate the costs of an order and prepare orders to be assembled
2       private class ProcessManufacturerOrderRequestsBehaviour extends Behaviour{
3           private boolean allOrdersProcessed = false;
4           private PredictedOrderInformation predictedOrderInformation = new PredictedOrderInformation();
5           public void action() {
6               MessageTemplate mt = MessageTemplate.or(MessageTemplate.MatchPerformative(ACLMessage.REQUEST), ↩
        MessageTemplate.MatchPerformative(ACLMessage.INFORM));
7               ACLMessage msg = myAgent.receive();
8               if(msg != null) {
9                   try {
10                      // Transform strings to java objects
11                      ContentElement ce = getContentManager().extractContent(msg);
12                      if(ce instanceof Action) {
13                          Concept action = ((Action)ce).getAction();
14                          if(action instanceof CalculateOrderCost) {
```

```
15                          CalculateOrderCost orderCost = (CalculateOrderCost) action;
16                          // Create predicate
17                          PredictedOrderCost predictedOrderCost = new PredictedOrderCost();
18                          // Store predicted order information in case that the manufacturer accepts the order so the calculation is only ↩
    done once
19                          predictedOrderInformation = calculateOrderInformation(orderCost.getOrder());
20                          predictedOrderCost.setCost(predictedOrderInformation.getMinimumPredictedCost());
21                          // Send to the manufacturer a predicted cost for the order
22                          ACLMessage msgOrderCost = new ACLMessage(ACLMessage.INFORM);
23                          msgOrderCost.addReceiver(manufacturerAID);
24                          msgOrderCost.setLanguage(codec.getName());
25                          msgOrderCost.setOntology(ontology.getName());
26                          try {
27                              // Transform java objects to strings
28                              getContentManager().fillContent(msgOrderCost, predictedOrderCost);
29                              myAgent.send(msgOrderCost);
30                          } catch (CodecException codece) {
31                              codece.printStackTrace();
32                          } catch (OntologyException oe) {
33                              oe.printStackTrace();
34                          }
35
36                      } else if(action instanceof PrepareOrderToAssemble) {
37                          // Add order to pending orders
38                          PendingOrder pendingOrder = new PendingOrder();
39                          pendingOrder.setOrder(predictedOrderInformation.getOrder());
40                          pendingOrder.setDaysLeftToAssemble(predictedOrderInformation.getPredictedAssemblytime());
41                          pendingOrders.add(predictedOrderInformation.getPredictedPositionInPendingOrdersList(), pendingOrder);
42                          // To minimise storage costs, when there are more than one supplier to order components from, different ↩
    orders of supplies for different days are going to be made
43                          int quantity = predictedOrderInformation.getOrder().getQuantity();
44                          int time = predictedOrderInformation.getPredictedAssemblytime();
45                          for(Map.Entry<Supplier, ArrayList<Component>> entry : predictedOrderInformation.↩
    getComponentsToOrderFromSuppliers().entrySet()) {
46                              SuppliesNeeded suppliesNeeded = new SuppliesNeeded(entry.getKey(), entry.getValue(), quantity, time);
47                              suppliesToBeOrdered.add(suppliesNeeded);
48                          }
49                      }
50                  } else if(ce instanceof NoMoreOrdersToday) {
51                      allOrdersProcessed = true;
52                  }
53              } catch (CodecException ce) {
54                  ce.printStackTrace();
55              } catch (OntologyException oe) {
56                  oe.printStackTrace();
57              }
58          } else {
59              block();
60          }
61      }
62
63      public boolean done() {
64          return allOrdersProcessed;
65      }
66  }
67
```

Listing 15: Warehouse RequestSuppliesToSuppliersBehaviour

```
1   // Behaviour that is going to request supplies to the suppliers
2   private class RequestSuppliesToSuppliersBehaviour extends OneShotBehaviour{
3       public void action() {
4           // Minimise warehouse expenses by making sure that all the supplies for an order arrive the same day
5           ArrayList<SuppliesNeeded> suppliesToRemove = new ArrayList<SuppliesNeeded>();
6           for(SuppliesNeeded supplies : suppliesToBeOrdered) {
7               // Check if the supplies should be ordered today
8               if(supplies.getTimeLeftToRequestDelivery() == supplies.getSupplier().getDeliveryTime()) {
9                   // Create agent action
10                  SellSupplies sellSupplies = new SellSupplies();
11                  Supplies s = new Supplies();
12                  s.setComponents(supplies.getComponents());
13                  s.setComponentsQuantity(supplies.getQuantityPerComponent());
14                  sellSupplies.setSupplies(s);
15                  // Create wrapper
16                  Action request = new Action();
17                  request.setAction(sellSupplies);
18                  request.setActor(supplies.getSupplier().getSupplierAID());
19                  // Send message to supplier requesting the supplies
20                  ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
21                  msg.addReceiver(supplies.getSupplier().getSupplierAID());
22                  msg.setLanguage(codec.getName());
```

```
                    msg.setOntology(ontology.getName());
                    try {
                        // Transform java objects to strings
                        getContentManager().fillContent(msg, request);
                        myAgent.send(msg);
                    } catch (CodecException ce) {
                        ce.printStackTrace();
                    } catch (OntologyException oe) {
                        oe.printStackTrace();
                    }
                    // Get the cost of the supplies and add it to the daily supplies purchased cost
                    for(Component component : supplies.getComponents()) {
                        dailySuppliesPurchasedCost = component.getPrice() * supplies.getQuantityPerComponent();
                    }
                    // Add supplies to the list of supplies to remove
                    suppliesToRemove.add(supplies);
                } else {
                    // Decrease the time left to request delivery
                    supplies.setTimeLeftToRequestDelivery(supplies.getTimeLeftToRequestDelivery() - 1);
                }
            }
            // remove supplies
            suppliesToBeOrdered.removeAll(suppliesToRemove);
        }
    }

```

Listing 16: Warehouse SendReadyOrdersBehaviour

```
    // Behaviour that is going to send orders ready to the manufacturer
    private class SendReadyOrdersBehaviour extends OneShotBehaviour{
        public void action() {
            // Create predicate
            OrdersReady ordersToSend = new OrdersReady();
            ordersToSend.setOrders(readyOrders);
            // Send message
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.addReceiver(manufacturerAID);
            msg.setLanguage(codec.getName());
            msg.setOntology(ontology.getName());
            try {
                // Transform java objects to strings
                getContentManager().fillContent(msg, ordersToSend);
                myAgent.send(msg);
            } catch (CodecException ce) {
                ce.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
            // Clear list
            readyOrders.clear();
        }
    }

```

Listing 17: Warehouse SendDailyWarehouseExpensesBehaviour

```
    // Behaviour that is going to send the daily warehouse expenses to the manufacturer when asked
    private class SendDailyWarehouseExpensesBehaviour extends Behaviour{
        int step = 0;
        public void action() {
            switch(step) {
            // Wait for query message from manufacturer
            case 0:
                MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF);
                ACLMessage msg = myAgent.receive(mt);
                if(msg != null) {
                    try {
                        // Transform strings to java objects
                        ContentElement ce = getContentManager().extractContent(msg);
                        if(ce instanceof WarehouseExpensesToday) {
                            step++;
                        }
                    } catch (CodecException ce) {
                        ce.printStackTrace();
                    } catch (OntologyException oe) {
                        oe.printStackTrace();
```

```
                }
            } else {
                block();
            }
            break;
        // Send daily warehouse expenses to the manufacturer
        case 1:
            // Create predicate
            WarehouseExpenses expenses = new WarehouseExpenses();
            expenses.setPenaltiesCost(dailyPenaltiesCost);
            expenses.setStorageCost(dailyStorageCost);
            expenses.setSuppliesCost(dailySuppliesPurchasedCost);
            // Send expenses
            ACLMessage response = new ACLMessage(ACLMessage.INFORM);
            response.addReceiver(manufacturerAID);
            response.setLanguage(codec.getName());
            response.setOntology(ontology.getName());
            try {
                // Transform java objects to strings
                getContentManager().fillContent(response, expenses);
                myAgent.send(response);
            } catch (CodecException ce) {
                ce.printStackTrace();
            } catch (OntologyException oe) {
                oe.printStackTrace();
            }
            step++;
            break;
        }
    }

    public boolean done() {
        return step == 2;
    }
}

```

Listing 18: Battery constraints

```
public class Battery extends Component{

    private String charge;

    @Slot(mandatory = true, permittedValues = {"2000", "3000"})
    public String getCharge() {
        return charge;
    }
}

```

Listing 19: Ram constraints

```
public class Ram extends Component{

    private String amount;

    @Slot(mandatory = true, permittedValues = {"4","8"})
    public String getAmount() {
        return amount;
    }
}

```

Listing 20: Screen constraints

```
public class Screen extends Component{

    private String size;

    @Slot(mandatory = true, permittedValues = {"5","7"})
    public String getSize() {
        return size;
    }
}

```

Listing 21: Storage constraints

```
public class Storage extends Component{

    private String capacity;

    @Slot(mandatory = true, permittedValues = {"64", "256"})
    public String getCapacity() {
        return capacity;
    }
}

```

Listing 22: Base smartphone constrains

```
public class Smartphone implements Concept{

    private Screen screen;
    private Storage storage;
    private Ram ram;
    private Battery battery;

    @Slot(mandatory = true)
    public Screen getScreen() {
        return screen;
    }

    @Slot(mandatory = true)
    public Storage getStorage() {
        return storage;
    }

    @Slot(mandatory = true)
    public Ram getRam() {
        return ram;
    }

    @Slot(mandatory = true)
    public Battery getBattery() {
        return battery;
    }
}

```

Listing 23: Small smartphone constrains

```
public class SmallSmartphone extends Smartphone{

    // Small smartphones are always going to have a screen of size 5 and a battery of charge 2000
    public SmallSmartphone() {
        Screen screen = new Screen();
        screen.setSize("5");
        this.setScreen(screen);
        Battery battery = new Battery();
        battery.setCharge("2000");
        this.setBattery(battery);
    }
}

```

Listing 24: Phablet smartphones constrains

```
public class PhabletSmartphone extends Smartphone{

    // Phablet smartphones are always going to have a screen of size 7 and a battery of charge 3000
    public PhabletSmartphone() {
            Screen screen = new Screen();
            screen.setSize("7");
            this.setScreen(screen);
            Battery battery = new Battery();
            battery.setCharge("3000");
            this.setBattery(battery);
        }
}
```

```
13
```

Listing 25: Storage cost constrain

```
1       // Behaviour that is going to calculate the costs of the storage
2       private class CalculateStorageCostBehaviour extends OneShotBehaviour{
3           public void action() {
4               for(Map.Entry<Component,Integer> entry : stock.entrySet()) {
5                   dailyStorageCost = entry.getValue() * perComponentDailyStorageCost;
6               }
7           }
8       }
9
```

Listing 26: Shipping and assembly constrains

```
1       // Behaviour that is going to process pending orders
2       private class ProcessPendingOrdersBehaviour extends OneShotBehaviour{
3           public void action() {
4               // The pending orders need to be processed from first to last as they are reordered each time a new order is added to them
5               ArrayList<PendingOrder> ordersToRemove = new ArrayList<PendingOrder>();
6               for(PendingOrder pendingOrder : pendingOrders) {
7                   Order order = pendingOrder.getPendingOrder();
8                   // If the order required amount of smartphones cannot be assembled do not process any more orders
9                   if(numbSmartphonesAssembledToday + order.getQuantity() > maxNumberSmartphonesAssemblePerDay) {
10                      break;
11                  }
12                  // Check if the components needed are in stock
13                  Screen screen = order.getSmartphone().getScreen();
14                  Storage storage = order.getSmartphone().getStorage();
15                  Ram ram = order.getSmartphone().getRam();
16                  Battery battery = order.getSmartphone().getBattery();
17                  if(stock.containsKey(screen) && stock.containsKey(storage) && stock.containsKey(ram) && stock.containsKey(battery↩
    )) {
18                      // Check if there are enough components to assemble the order
19                      int screenAvailables = stock.get(order.getSmartphone().getScreen());
20                      int storageAvailables = stock.get(order.getSmartphone().getStorage());
21                      int ramAvailables = stock.get(order.getSmartphone().getRam());
22                      int batteryAvailables = stock.get(order.getSmartphone().getBattery());
23                      int requiredQuantity = order.getQuantity();
24                      if(screenAvailables >= requiredQuantity && storageAvailables >= requiredQuantity && ramAvailables >= ↩
    requiredQuantity && batteryAvailables >= requiredQuantity) {
25                          // Update number of smartphones assembled today
26                          numbSmartphonesAssembledToday += requiredQuantity;
27                          // Add order to ready orders list
28                          readyOrders.add(order);
29                          // Update stock
30                          stock.replace(screen, screenAvailables - requiredQuantity);
31                          stock.replace(storage, storageAvailables - requiredQuantity);
32                          stock.replace(ram, ramAvailables - requiredQuantity);
33                          stock.replace(battery, batteryAvailables - requiredQuantity);
34                          // Remove current order from pending orders
35                          ordersToRemove.add(pendingOrder);
36                      } else {
37                          // If there are not enough components of any type to assemble the current order do not process any more orders
38                          break;
39                      }
40                  } else {
41                      // If the components needed to assemble the current order are not in stock do not process any more orders
42                      break;
43                  }
44              }
45              // Remove ready orders from pending orders
46              if(!ordersToRemove.isEmpty()) {
47                  pendingOrders.removeAll(ordersToRemove);
48              }
49          }
50      }
51
```

Listing 27: Penalty constrain

```
1       // Behaviour that is going to update the due date and left days to assembly of the pending orders that have not been assembled ↩
        today and calculate any penalties before adding new orders
2       private class UpdatePendingOrdersTimesAndCalculatePenaltiesBehaviour extends OneShotBehaviour{
3           public void action() {
4               for(PendingOrder pendingOrder : pendingOrders) {
5                   pendingOrder.setDaysLeftToAssemble(pendingOrder.getDaysLeftToAssemble() − 1);
6                   pendingOrder.getPendingOrder().setDueDate(pendingOrder.getPendingOrder().getDueDate() − 1);
7                   if(pendingOrder.getPendingOrder().getDueDate() < 0) {
8                       dailyPenaltiesCost += pendingOrder.getPendingOrder().getPenalty();
9                   }
10              }
11          }
12      }
13
```