Multi-Agent Systems Coursework

Contents

List of Figures

List of Tables

1 Introduction

Manufacturing companies hardly ever handle all the processes that are behind the production of a good, which span from the extraction of raw materials through to the delivery of the final product. Instead, they are commonly part of a supply chain: they buy goods, perform some work on these, and sell them to the next customer. In other words, each company does not only act as the manufacturer, but also as the customer and the supplier.

This project aimed at designing and implementing a PC supply chain multi-agent system in JADE; this system consists of three main separate agents: the customer, the manufacturer and the supplier, described in more detail in the next section. The project focuses on the manufacturer agent and the development of a strategy that maximises its profit. Moreover, particular attention was put into the implementation of an ontology and the effective use of communication protocols.

The dynamics of the enterprise market make the coordination for this type of system uncertain: materials delivery can be delayed, production can experience downtime, customers can cancel orders, and other issues can cause deviations from the plan (Um *et al.* , 2010). Agents must react and adapt to these changes, making optimal decisions based on complex global criteria that might not be entirely known and may involve conflicting interests. Each agent must actively participate in the collaboration process: they need to determine the effect that their decisions will have on other agents and coordinate with them to choose an alternative solution that is optimal for the entire supply chain.

Multi-agent systems (MAS) reveal extremely suitable for this type of problem: agents possess features that can power autonomous, adaptive, collaborative and intelligent systems that allow companies to automate their inter-processes, such as providing automatic partner discovery, interaction and coordination, as well as their intra-processes, such as managing the production schedule and stock levels.

2 Model design

2.1 The Agents

The system is composed by four agent types: the customer, the manufacturer, the supplier and the ticker. These are described below:

- Customer - on each day, it generates one order for a number of PCs of an identical, randomly generated specification. If the order is accepted and finally received, the customer sends a payment to the manufacturer.
- Manufacturer - the manufacturer accepts or declines customer orders, orders components from the suppliers, assembles the components into PCs and sends the completed orders to the customers. It also makes payments to the suppliers and receives them from customers. Its aim is to maximise the profit it makes for each PC order.

- Supplier - each supplier has different speeds and delivery times, and owns unlimited computer components. Suppliers accept components orders and ship them to the manufacturer. They receive payments for the components orders once these are made.
- Ticker - the ticker agent keeps track of the passage of time (expressed in days) and synchronises the other agents via the transmission of messages.

2.2 The Ontology

An ontology is a formal representation of knowledge. It provides machine-interpretable definitions of basic concepts and relations among them (Noy *et al.* , 2001). This allows software agents to share a common understanding of the structure of information and inter-operate with other agents.

A supply chain specific ontology was created to describe the concepts, predicates and actions that agents can use to understand the world. In Jade, these elements are defined as classes and can be encoded within messages in a standard FIPA format. Agents are then able to share these messages with other agents, that decode them on receipt.

2.2.1 Concepts

Concepts represent entities or "things" in the world, and often correspond to nouns or noun phrases. Concepts may also share relationships with each other. The concepts implemented for the supply chain domain are illustrated in Appendix A.1, beginning with the elements at the lowest levels of the hierarchy. Design decisions are justified below.

All the components in the system inherit from the ComputerComponent concept (see Figure 3). It was implemented as an abstract class to allow agents to exchange information about any of the specific components in a generalised way (e.g. loop through all the components of an order thanks to the data structure `ArrayList<ComputerComponent>`).

The same principles apply for the components Cpu, Motherboard and OS (Figure 3), subclassed by their platform specif versions (desktop or laptop). Thanks to their abstract nature, it is possible to exhange information about these components and always expect their existence in a computer, regardless of their concrete implementation.

It may be argued that the HardDrive and Ram classes could also specify subclasses that distinguish on storage capacity, however, this difference was deemed not necessary for the purposes of this simulation.

The Computer abstract class defines a structure common to all computers, as shown in Figure 4. The Order class contains a a Computer reference (see Figure 5); this allows flexible information exchange: a customer order is always expected to contain a computer, it is of no importance whether this is a laptop or a desktop. The efficacy of this design is evident when dealing with laptops, substantially different from desktops since they also have a screen: the manufacturer can simply call the Order `getComputer()` method and get the laptop instance that was abstracted by the Computer class.

2.2.2 Predicates and Actions

According to the FIPA-SL standard (FIPA, 2002), when an agent asks or informs another agent about the truth value (or state) of something in the world, the content of the message must be of type Predicate, whereas when an agent requests another agent to perform a task, it must be of type Action. All the predicates and actions created for the supply chain system are illustrated and described respectively in Figure 6 and Figure 7.

The predicates CanManufacture and OwnsComponents are used to ensure that the receiving agents will be able to carry out, respectively, the actions MakeOrder and BuyComponents before requesting them.

The action AskSuppInfo is used to get the supplier's list of prices and delivery time, which are then sent back with the SendsSuppInfo predicate. The manufacturer uses this information to calculate the most profitable way to complete an order (see section 4).

The predicates ShipComponents and ShipOrder are used to deliver the requested goods to the initiator agents. When these are received, the agents send back a payment with the SendsPayment predicate.

2.3 Communication Protocols

A sequence diagram of the full system is shown in Figure 8.

The FIPA Request Interaction Protocol (IP) specification states that when an agent wants another agent to perform a certain task, this needs to be communicated with a REQUEST performative. This protocol is used by customers to request a manufacturer to make an order (Figure 9) and by the manufacturer to request the supplier to send components (Figure 10). The specification states that the response of a request message that carries some result should be sent using the INFORM-RESULT performative, however, this is not included in JADE, so a general INFORM was deemed the most appropriate option.

The customer and the manufacturer are designed to act cautiously: before sending requests to make an order or buy components, they send QUERY_IF messages to ask for the availability of the receiving agents to carry out these actions (Figures 9 and 10). This was deemed a desirable addition as the real world agents might be unavailable (e.g. the manufacturer does not accept more orders due to excessive workload or the supplier does not own the needed components). Although the FIPA specification states that the response of a QUERY_IF interaction should be sent using the INFORM-true/false performative to indicate positive or negative outcomes, the system uses the performatives CONFIRM and DISCONFIRM, considered more semantically appropriate responses for the provided predicates.

For the manufacturer to get components, the Contract Net Protocol was considered, however, this would have introduced greater complexity in the system and exponentially increased the number of messages passed. For these reasons, it was decided to retrieve prices and delivery times only once at the start of each day using the FIPA request IP (Figure 10).

3 Model implementation

3.1 Agents Behaviours

Table 1: Customer behaviours

Name	Type	Description
TickerWaiter	cyclic	Behaviour that synchronises the agent with the global timing system.
ReceiveOrder	cyclic	Listens for incoming completed orders. Cyclic because it is not sure when to expect the orders, as they could be delayed.
DailyActivity	sequential	Allows to run all the behaviours listed below in a sequential way.
CreateOrder	one shot	Creates a random order.
FindManufacturers	one shot	Looks for manufacturers on the yellow pages.
AskIfCanManufacture	one shot	Asks the manufacturer if it can manufacture the order.
MakeOrderAction	custom	It repeats for the number of QUERY_IFs sent. If the manufacturer replies positively, it sends a request message back.
EndDay	one shot	Sends a done message to the Ticker.

Table 2: Manufacturer behaviours

Name	Type	Description
TickerWaiter	cyclic	Behaviour that synchronises the agent with the global timing system.
DailyActivity	sequential	Allows to run all the behaviours listed below in a sequential way.
FindCustomers	one shot	Looks for customers on the yellow pages.
FindSuppliers	one shot	Looks for suppliers on the yellow pages.
GetInfoFromSuppliers	multi-step	Sends requests for info about the prices and delivery time to the suppliers.
OrderReplyBehaviour	custom	Accepts or declines a customer order depending on the profit it will make. It repeats until all customers have asked once.
ManageOrderRequests	multi-step	Accepts the requests for the orders approved in the previous behaviour. Asks the most convenient supplier if they own the components quantities needed. If the response is positive, it sends a request message for the components and finally sends a payment.
ReceiveComponents	custom	Loops for all the component orders scheduled to arrive today and receives them.
ManufactureAndSend	multi step	Assembles the order and sends it to the customer. Subsequently, it receives the payments from the customers.
EndDay	one shot	Calculates the profit. Sends a done message to the Ticker and the suppliers.

Table 3: Supplier behaviours

Name	Type	Description
TickerWaiter	cyclic	Behaviour that synchronises the agent with the global timing system.
SendComponents	one shot	Send the components when the set supplier delivery days have passed.
FindBuyers	one shot	Looks for manufacturers on the yellow pages.
ReplySuppInfo	cyclic	Sends the supplier's info (prices and delivery days) to the manufacturer.
OffersServer	cyclic	Replies to the messages asking if the supplier owns a set number of components.
ReceiveRequests	cyclic	Receives requests for component orders.
ReceivePayment	cyclic	Receives the payment for the manufacturer orders.
EndDayListener	cyclic	Waits for the manufacturer's done message and sends one to the Ticker.

3.2 System Constraints

Different components for desktops and laptops - this constraint is enforced by the use of the concrete classes Laptop and Desktop, that inherit from the Computer abstract class. As shown in the code snippets 2, 3 and 4, Laptop and Desktop inherit the common components Cpu, Motherboard, Ram, HardDrive and Os, from Computer. They then use *variable hiding* to specify the type of Cpu and Motherboard. In addition, Laptop has a reference to a Screen.

The component delivery times are enforced with the aid of an order wrapper, shown in Listing 5. It holds a variable named "deliveryDay" that is assigned when an order is received. The supplier runs a one-shot behaviour daily in order to ship the orders that have a "deliveryDay" equal to the day count (see Listing 6).

Per-day warehouse storage cost - when the components are received by the manufacturer, these are put in the warehouse hashMap. At the end of the day, a simple for loop checks how many components it contains, multiplies it by 50 and detracts that amount from the daily profit (see Listing 7).

Enough components to build order - after receiving new components from the suppliers, the manufacturer checks that it has enough of them in the warehouse to assemble and send the flagged customer orders (see Listing 8). If it does not, this task is tried again the following day.

Penalties for late delivery - when a customer order is received, it is wrapped in a helper object and the receiving date is logged. At the end of the day, the manufacturer loops through the orders and for each it calls a method returning the exact delivered by date. If this date is equal or precedes today's date, the order is late and a fee is applied. See Listing 9 for a demonstration.

Total profit calculation - the penalty for late orders and the warehouse storage fees are calculated at the end of each day, whereas the total value of orders shipped and the supplies purchased are updated when their payments are received and made. See Listing 10.

4 Design of manufacturer agent control strategy

Listing 1: Manufacturer control strategy

```
1  Request supplier to send information about them (prices and delivery time)
2  Receive customer message asking if order can be manufactured
3  FOR each supplier {
4    //Calculate the cost of buying all components from this supplier
5    FOR components in order {
6      supplierCost += supplierPrices.getComponent(component) * order.getQuantity()
7    }
8    // Calculate which supplier will grant us the highest profit
9    lateDeliveryFee = 50 * days after orderDueDate the supplier would ship components
10   profit = value of order - supplierCost - lateDeliveryFee
11   IF profit > maxProfit && > 0 { save bestSupplier, supplierCost and maxProfit }
12 }
13 IF bestSupplier is not null {
14   save the best supplier in an Order wrapper instance, set state of order to APPROVED
       and reply with CONFIRM to customer
15 } ELSE { send DISCONFIRM to customer }
16
17 Receive order REQUEST from customer
18 Send QUERY_IF to best supplier asking if it owns the components needed for the order
19 IF CONFIRM is received { buy all components for order from best supplier via REQUEST }
20 ELSE { ask second best supplier, IF profit is still positive }
21 // n.b. else logic not implemented. Unlimited components for this simulation
22 Send payment to supplier and update paymentToSupplier
23
24 // Received components are assigned to a specific order; assemble and ship it
25 Receive components from supplier and set the order they are for to COMPS_RECEIVED
26 Store the components in the warehouse
27 FOR all order with state COMPS_RECEIVED {
28   IF enough components to make order { manufacture, ship and flag as AWAITING_PAYMENT }
29   ELSE { continue loop }
30 }
31 Receive payment and update var receivedPayments (money)
32
33 // Calculate daily profit and add to total profit
34 totalProfit += receivedPayments - paymentToSupplier - late order fees - warehouse
     storage
```

The manufacturer control strategy is presented above in the form of pseudocode, in a much more simplified way. The aim is to minimise the costs in order to maximise the profit. The highest cost comes from warehouse storage; the solution above completely eliminates this cost as there will never be spare components to store. The late delivery fee is negligible (see next section). Since the supplier prices and late delivery fee are known, the manufacturer can foresee the costs involved and always choose the supplier that grants the highest possible profit for that order (lines 5 to 8).

5 Experimental results

From early experimentation it was found that component pre-ordering causes the highest loss for the manufacturer. An order is composed by an average of 163 components and the daily storage fee for each component is £5. Even achieving a perfect accuracy in predicting which components to buy in advance, these would provide an average daily loss of £815 per order. Hence, to maximise the manufacturer's profit several strategies that avoided this fee were attempted. These, in chronological order are:

1. **Time constrained strategy**. Order the components from the supplier that can deliver them within the customer set due date at the cheapest price. This strategy does not involve pre-ordering nor attempts late delivery.
2. **Cheapest supplier strategy**. Always order the components from the cheapest supplier, regardless of the late delivery fees. It was established that the daily late delivery fee (£50), in comparison to the average daily loss caused by storage fees (£815), was negligible and led to a higher profit thanks to the lower component price offered by this supplier.
3. **Smarter strategy**. For each supplier, calculate in advance the costs that an order will incur. This strategy accounts for the total component cost and the total late delivery fee that an order would experience, and picks the most profitable solution among the ones offered by the three suppliers. Moreover, only the orders that result remunerative at the end of this calculation are accepted. See Listing 10 for an example of its implementation.

These control strategies were run for 90 days, at the end of which their profit was logged; each strategy was run a total of 30 times. Their results in terms of *pure* profit[†] are presented in Figure 1, whereas Figure 2 shows the Standard Deviation (SD) and Standard Error of the Mean (SEM).

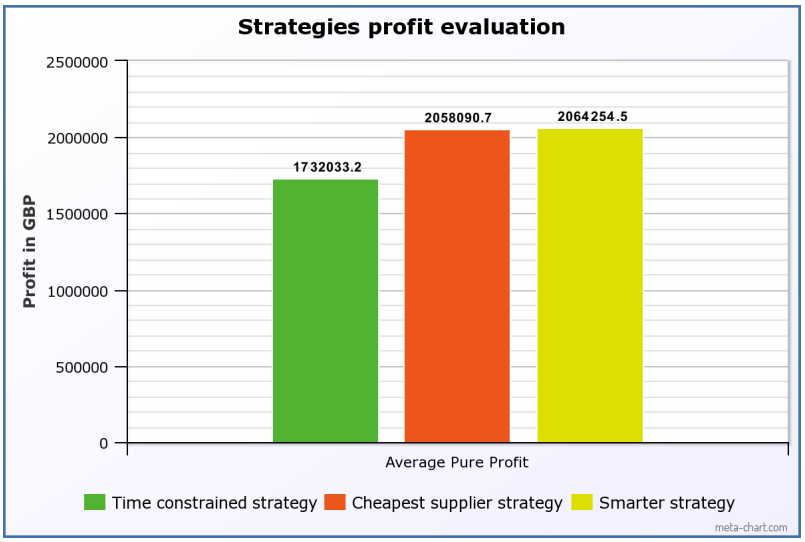

Figure 1: The time constrained strategy performed worst, with a gap of about £320,000 from the other two. The cheapest supplier strategy performed similarly to the smarter strategy, with a gap of only £6,163.7.

[†]To calculate the pure profit, the manufacturer was set to refuse any order after the 83rd day of the simulation; this is necessary to avoid paying for the components of many orders that are not cashed in after this date, which would provide an erroneous result.

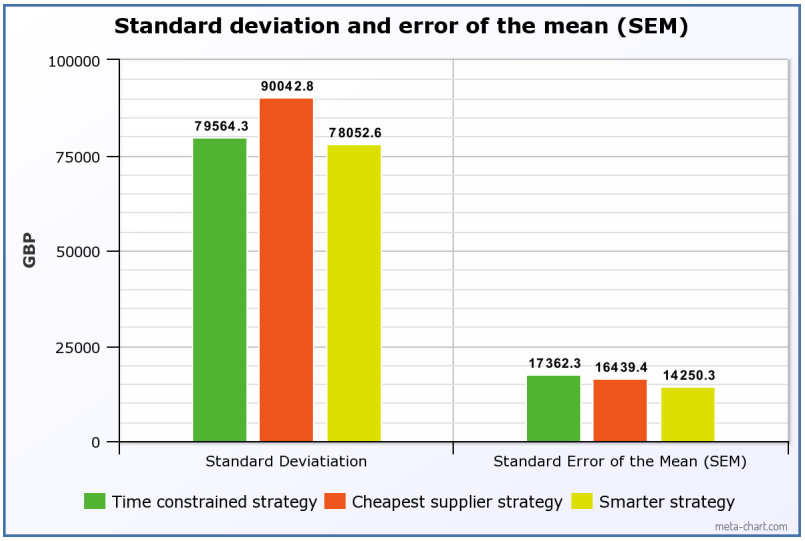

Figure 2: Shown at the side are the Standard Deviation (SD) and Standard Error of the Mean (SEM). The SD is similar for the first and third strategies, while the cheapest supplier strategy performs worst. The SEM is similar for all three strategies, although the first strategy performs worst. The smarter strategy achieves better results for both calculations.

Increasing the number of customers or the price of the warehouse storage fee does not affect the result tendency shown above.

6 Conclusion

In conclusion, the manufacturer *smarter* control strategy performs extremely well, achieving a high profit during the course of 90 days, as illustrated in Figure 1. The implemented strategy also grants consistent results across trials, as evidenced in Figure 2. To the current knowledge, the system could not achieve a higher profit than the one tested. From another stand point, however, this strategy would not be feasible for a real system: the maximum profit is achieved taking advantage of the low late delivery fees, which could not work over extended periods due to reputation involvement.

The system works on a well designed ontology, which allow the agents to exchange information in a robust and consistant way. This could be further improved with the introduction of a JADE vocabulary.

The codebase is split into appropriate behaviours that are of appropriate specificity and easy to track, while the system is simple to understand. The agents communication is of adequate standard and complies to the FIPA Request and Query IP specifications. Moreover, attempts were made to ensure the agents would be capable of performing the desired tasks before requesting them. On the other hand, to fully comply with the FIPA specification the ContractNet protocol could be implemented for buying the components from the supplier, although this would cause further system complexity and exponentially increase the number of messages passed between agents, as explained in section 2.3.

References

FIPA. 2002. *FIPA SL Content Language Specification*. http://www.fipa.org/specs/fipa00008/SC00008I.html#_Toc26668865. Accessed: 2018-12-3.

Lhotka, Rockford. 2005. *Should all apps be n-tier?* http://www.lhotka.net/weblog/ShouldAllAppsBeNtier.aspx. Accessed: 2018-11-10.

Noy, Natalya F, McGuinness, Deborah L, *et al.* . 2001. *Ontology development 101: A guide to creating your first ontology.*

Um, Wansup, Lu, Huitian, & Hall, Teresa. 2010. A Study of Multi-Agent Based Supply Chain Modeling and Management. **2**(01), 333–341.

Appendices

A Ontology

Below are described all the AgentAction, Concept, and Predicate subclasses created and their hierarchical relations. Each class figure lists its slots and the slots types. All slots, indicated in the schemas with red squares, are mandatory unless otherwise stated.

A.1 Concepts

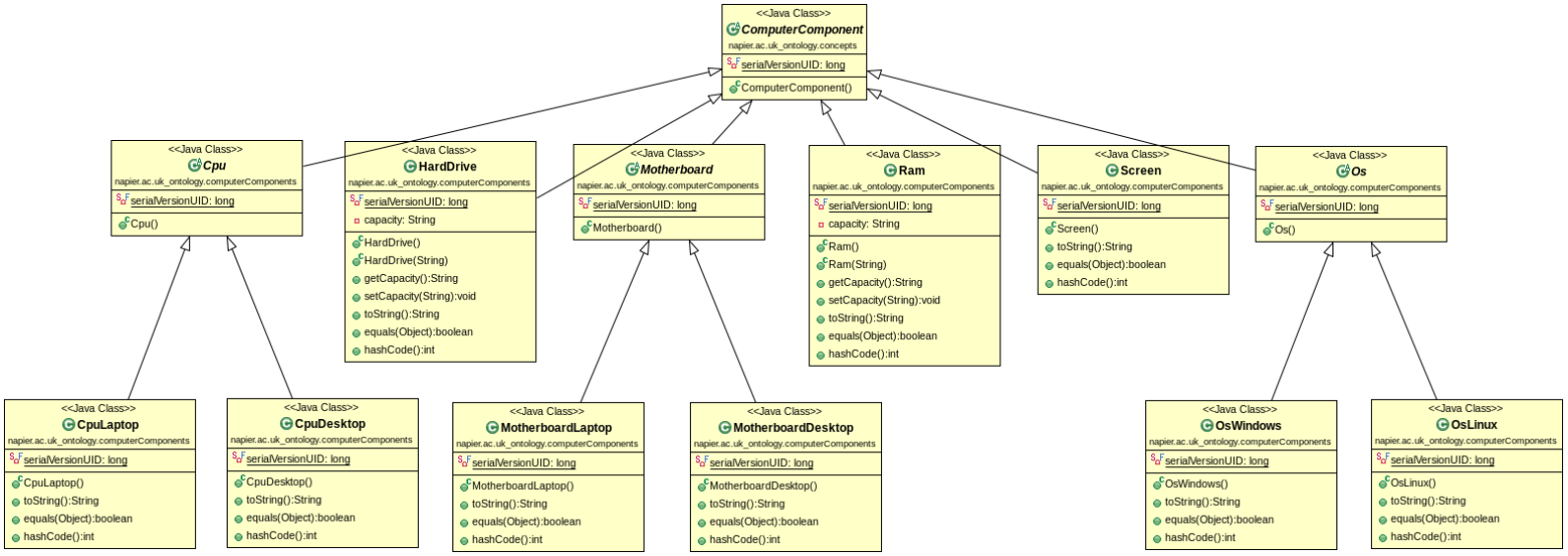

Figure 3: The ComputerComponent abstract class is a concept mapping the component of a computer and inherits from the Jade Concept class.

All the computer components in the system inherit from ComputerComponent and further divide into abstract and concrete classes. As illustrated in above, these are HardDrive, Ram and Screen (concrete classes), and the generic Cpu, Motherboard and OS (abstract classes).

The Cpu and Motherboard classes concretise in their desktop and laptop subclass versions, while the OS class concretises in Windows and Linux.

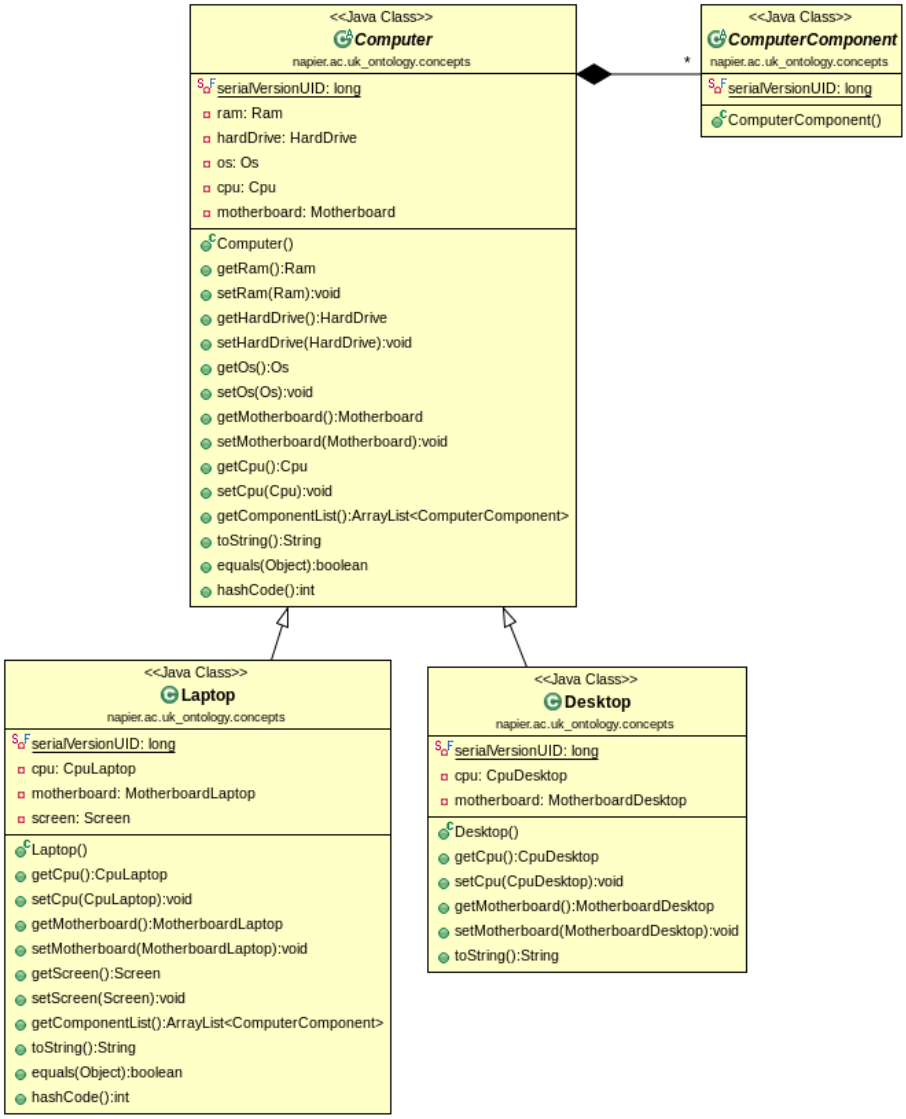

Figure 4: Higher in the hierarchy is the Computer abstract class; this concept abstracts a computer, which can either be a desktop or a laptop. The Computer class defines a structure common to computers: it holds references for the concrete classes ram and hard drive, and references of the generic classes OS, cpu and motherboard.

The Desktop and Laptop concrete subclasses inherit the concrete members from the Computer concept and, thanks to a technique named "variable hiding", they define the more specific types of Cpu and Motherboard they are composed of. In addition, the Laptop class contains a reference of the Screen class.

The Computer class is essentially composed of a group of computer components. The relation between this class and the ComputerComponent class is shown at the top-right of the schema.

In the above classes, all the slots are mandatory. `getComponentList()` is a commodity method used by the agents to retrieve a list of components in the form of an `ArrayList`.

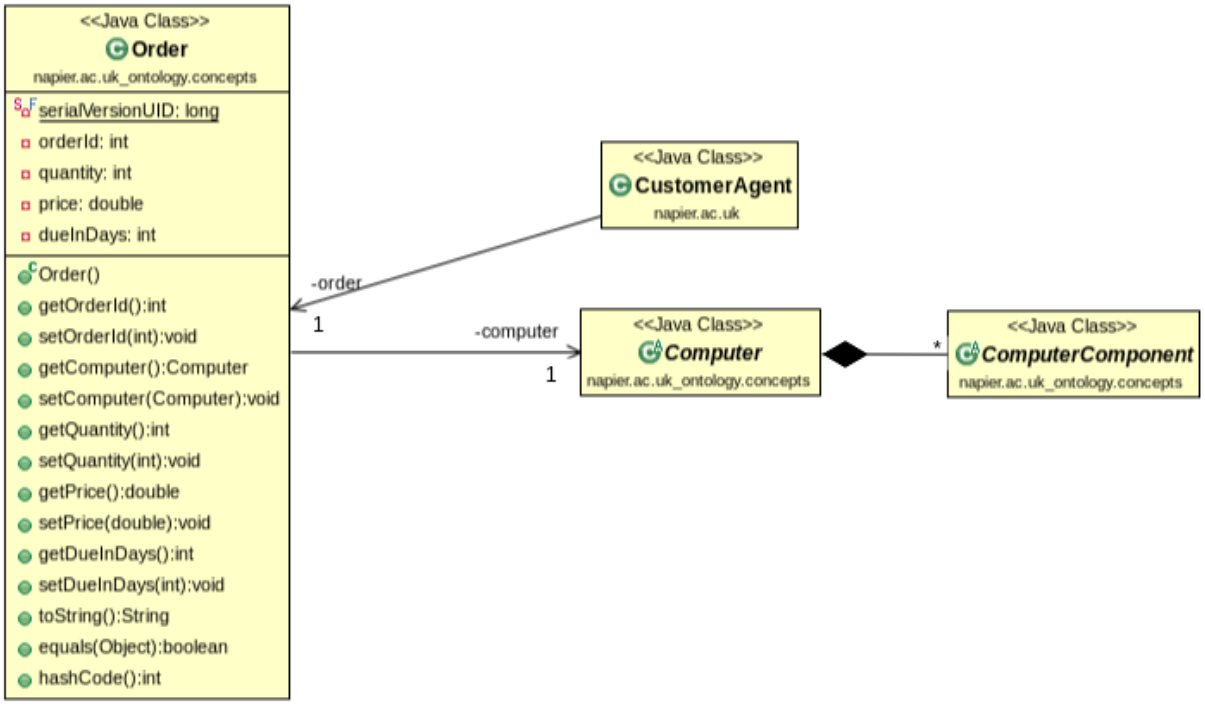

Figure 5: At the top of the hierarchy is the Order concept. It maps a customer order and contains a reference to the Computer concept, a quantity, a due date and a price. Each customer agent creates order object on each day.

A.2 Predicates

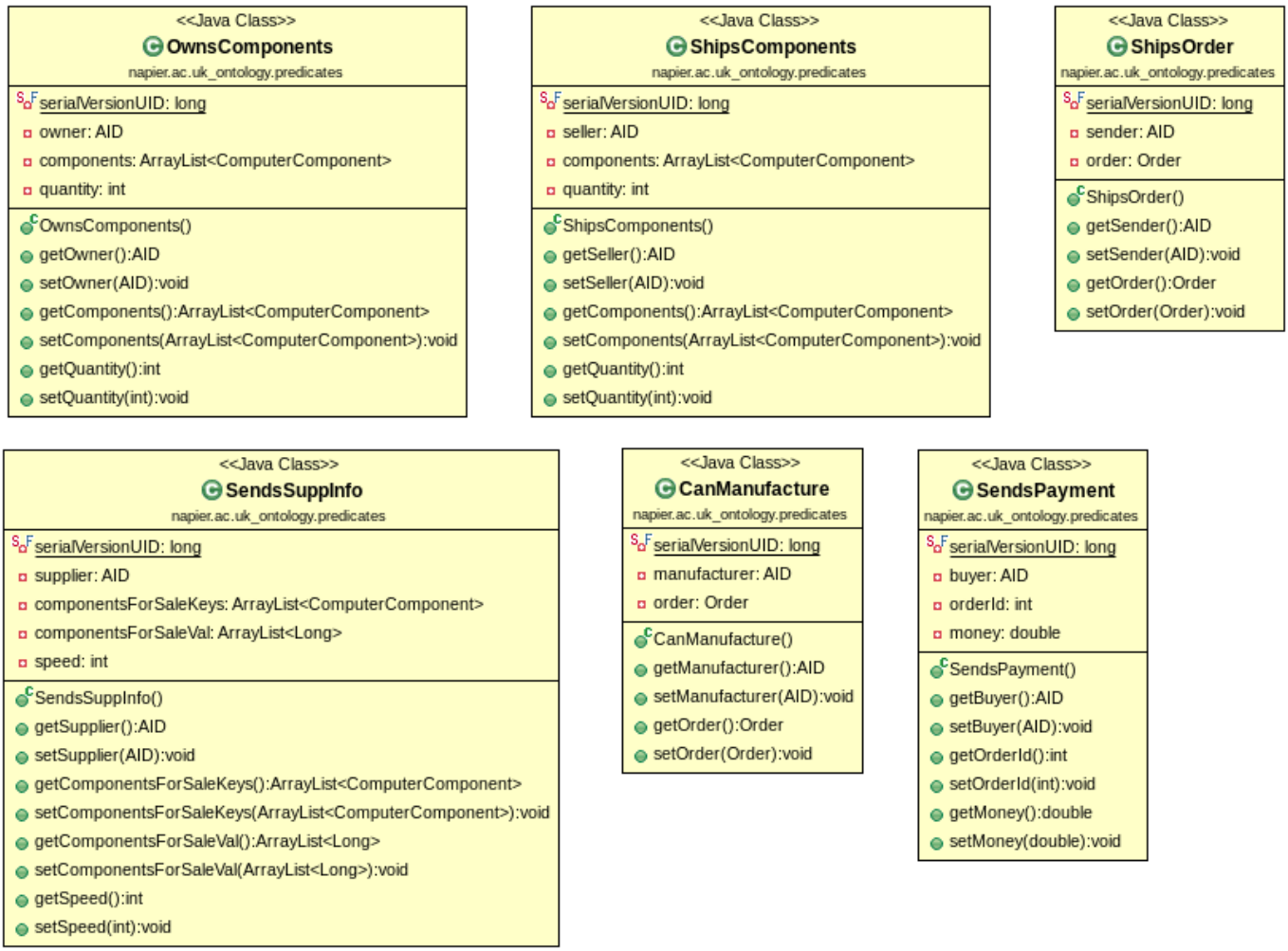

Figure 6: Below are the descriptions of each of the predicates included in the system.

- CanManufacture - asks a manufacturer if they can manufacture an order.
- OwnsComponents - asks a supplier if they own the quantity of components specified.
- SendsPayment - sends the payment of some sum.
- SendsSuppInfo - sends information regarding the supplier's prices and delivery times.
- ShipsComponents - sends the components to the manufacturer.
- ShipsOrder - sends the completed order to the customer.

To note is that, for sending information about the supplier's prices, it was not possible to use a hashmap in a slot; this had to be split into two lists.

A.3 Actions

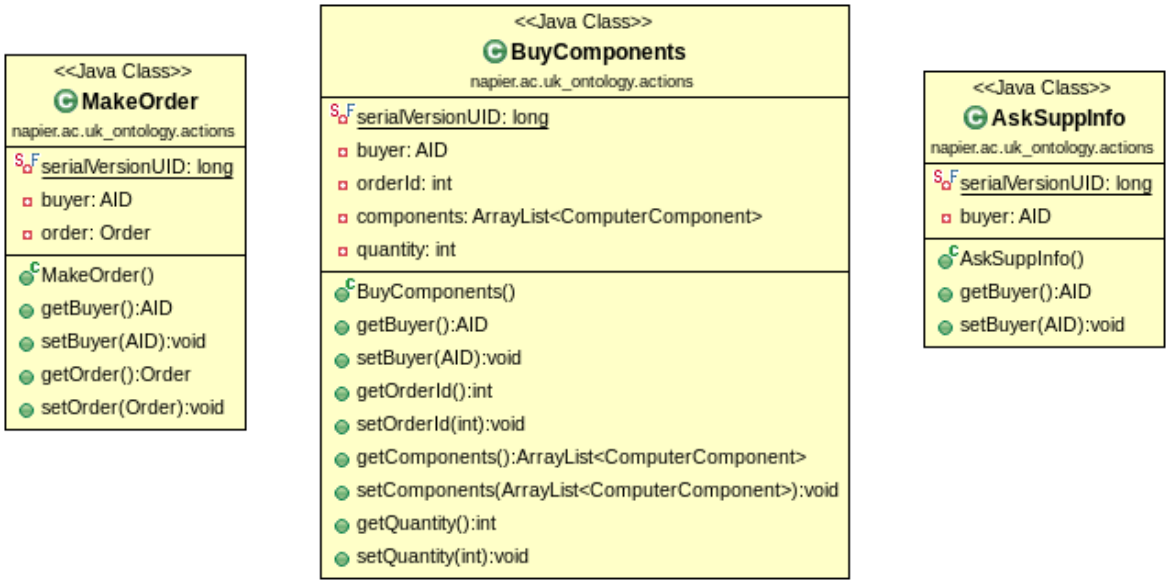

Figure 7: Below are the descriptions of each of the actions included in the system.

- MakeOrder, used by a customer to request a manufacturer to fulfil an order; it contains an Order object and a buyer AID.
- BuyComponents, used by the manufacturer to request a supplier to ship computer components; it contains a list of computer components, a quantity and a buyer AID.
- AskSuppInfo, used by the manufacturer to request a supplier to send information about their prices and their delivery times. It only contains a buyer AID.

B Communication Protocols

B.1 Sequence diagrams

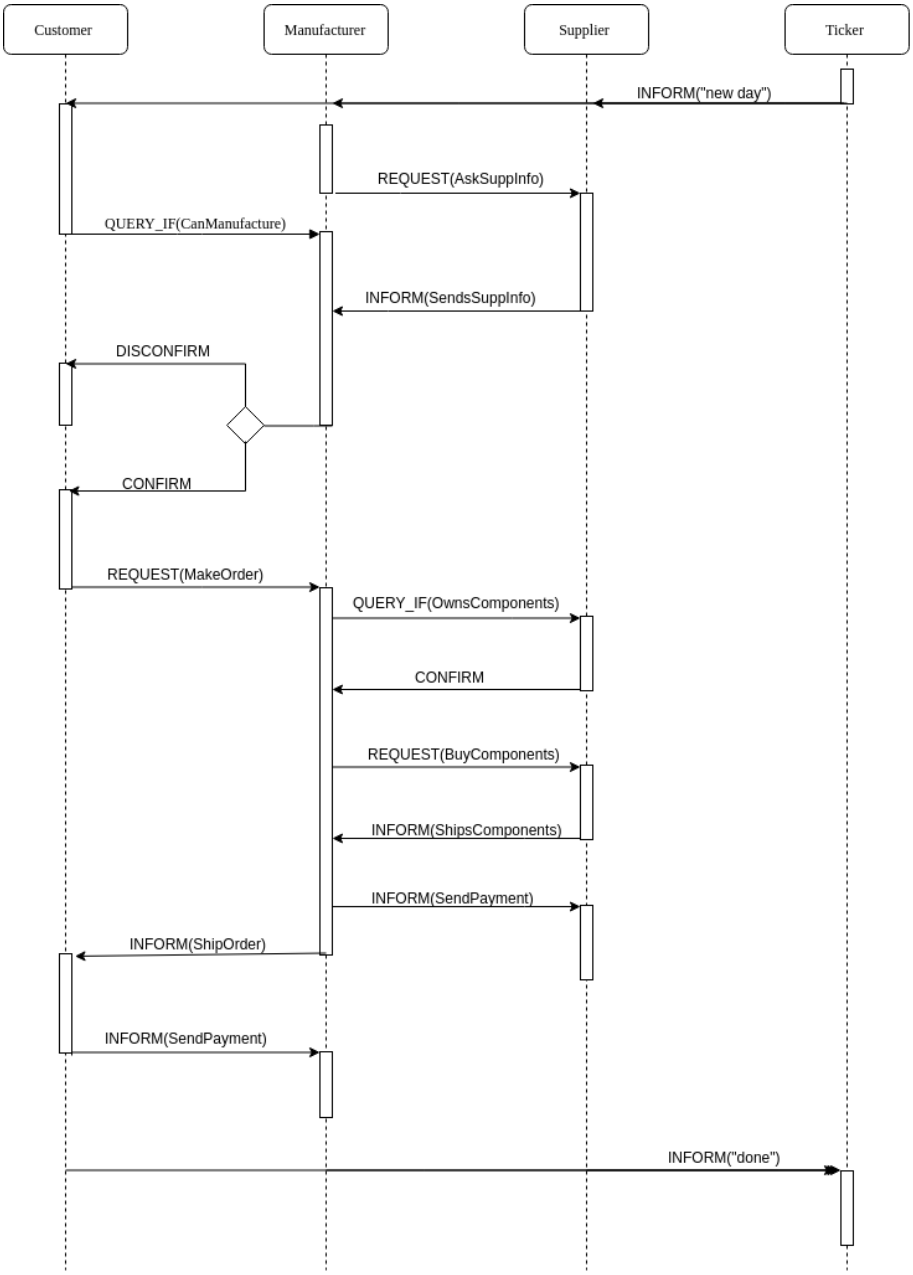

Figure 8: Above is illustrated the sequence that the agents follow to exchange information. Each message is accompanied by a performative. REQUESTs contain AgentActions, whereas INFORM and QUERY_IF messages contain predicates (apart from the ones to and from the ticker). For simplicity purposes, the messages exchanged with the Directory Facilitator (DF) to find the other agents in the system have been omitted.
For an example of the messages contents see section B.2.

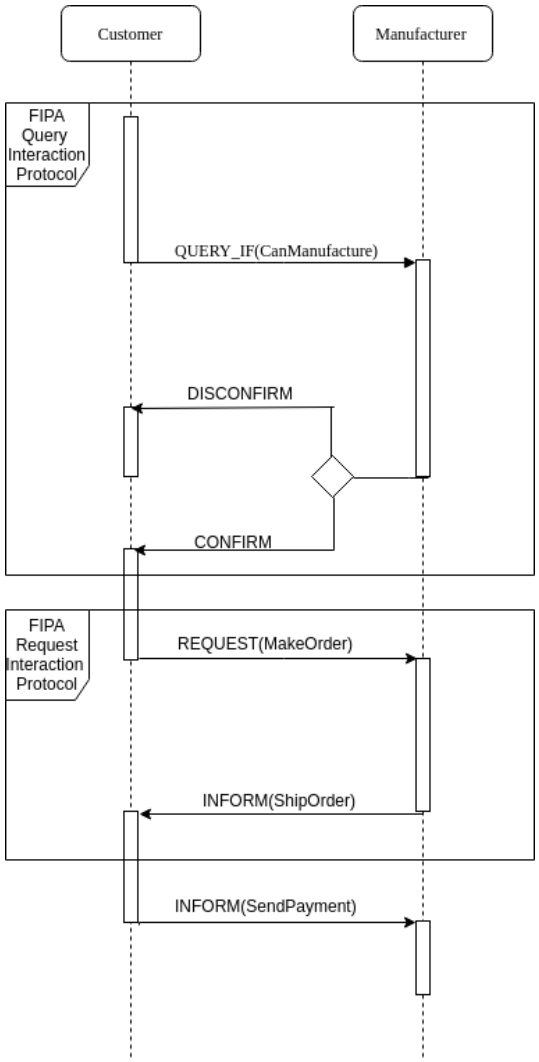

Figure 9: Show above are the communication protocols used by the customer and the manufacturer in an isolated way.

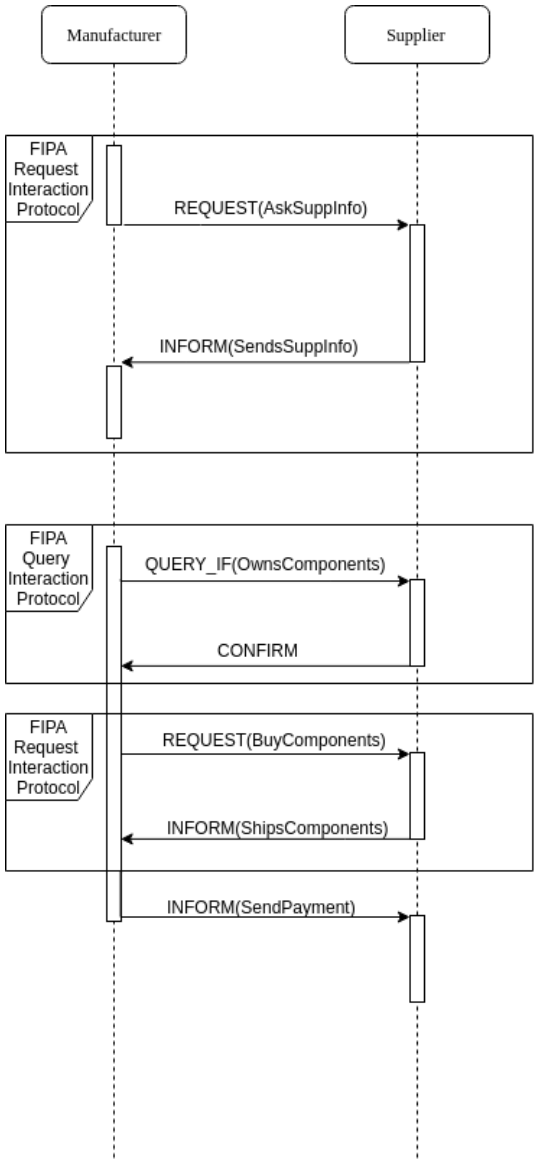

Figure 10: Show above are the communication protocols used by the manufacturer and the supplier in an isolated way.

B.2 Example message content

Below is shown the example content for each of the messages agents exchange:

- REQUEST(AsksSuppInfo)

```
((action
    (agent-identifier
     :name supplierSlow@192.168.0.2:1099/JADE
     :addresses (sequence http://francesco-pc:7778/acc))
    (AskSuppInfo
     :buyer
        (agent-identifier
         :name manufacturer@192.168.0.2:1099/JADE
         :addresses (sequence http://francesco-pc:7778/acc)))))
```

- INFORM(SendsSuppInfo)

```
((SendsSuppInfo
    (sequence (Ram
        :capacity "8GB") (OsWindows) (Screen) (CpuLaptop) (CpuDesktop) (Ram
        :capacity "16GB") (MotherboardLaptop) (OsLinux) (MotherboardDesktop)
            (HardDrive
        :capacity "1TB") (HardDrive
        :capacity "2TB"))
    (sequence 30 75 60 150 110 70 50 0 95 35 55) 7
    (agent-identifier
      :name supplierSlow@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc))))
```

- QUERY_IF(CanManufacure)

```
((CanManufacture
    (agent-identifier
      :name manufacturer@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc))
    (Order
      :computer
        (Laptop
          :cpu (CpuLaptop)
          :hardDrive
            (HardDrive
              :capacity "2TB")
          :motherboard (MotherboardLaptop)
          :os (OsLinux)
          :ram (Ram
              :capacity "16GB")
```

```
        :screen (Screen))
      :dueInDays 7
      :orderId -1
      :price 34272.0
      :quantity 48)))
```

- REQUEST(MakeOrder)

```
((action
    (agent-identifier
      :name manufacturer@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc))
    (MakeOrder
      :buyer
        (agent-identifier
          :name customer0@192.168.0.2:1099/JADE
          :addresses (sequence http://francesco-pc:7778/acc))
      :order
        (Order
          :computer
            (Laptop
              :cpu (CpuLaptop)
              :hardDrive
                (HardDrive
                  :capacity "2TB")
              :motherboard (MotherboardLaptop)
              :os (OsLinux)
              :ram (Ram
                  :capacity "16GB")
              :screen (Screen))
          :dueInDays 7
          :orderId -1
          :price 34272.0
          :quantity 48))))
```

- QUERY_IF(OwnsComponents)

```
((OwnsComponents
    (sequence (Ram
        :capacity "16GB") (HardDrive
        :capacity "2TB") (OsLinux) (CpuLaptop) (MotherboardLaptop) (Screen))
    (agent-identifier
      :name supplierSlow@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc)) 48))
```

- REQUEST(BuyComponents)

```
((action
    (agent-identifier
      :name supplierSlow@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc))
    (BuyComponents
      :buyer
        (agent-identifier
          :name manufacturer@192.168.0.2:1099/JADE
          :addresses (sequence http://francesco-pc:7778/acc))
      :components
        (sequence (Ram
            :capacity "16GB") (HardDrive
            :capacity "2TB") (OsLinux) (CpuLaptop) (MotherboardLaptop) (Screen))
      :orderId 1
      :quantity 48)))
```

- INFORM(ShipsComponents)

```
((ShipsComponents
    (sequence (Ram
        :capacity "16GB") (HardDrive
        :capacity "2TB") (OsWindows) (CpuDesktop) (MotherboardDesktop)) 3 45
    (agent-identifier
      :name supplierSlow@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc))))
```

- INFORM(ShipsOrder)

```
((ShipsOrder
    (Order
      :computer
        (Desktop
          :cpu (CpuDesktop)
          :hardDrive
            (HardDrive
              :capacity "2TB")
          :motherboard (MotherboardDesktop)
          :os (OsWindows)
          :ram (Ram
              :capacity "16GB"))
      :dueInDays 3
      :orderId 3
      :price 31320.0
      :quantity 45)
    (agent-identifier
      :name manufacturer@192.168.0.2:1099/JADE
```

```
      :addresses (sequence http://francesco-pc:7778/acc))))
```

- INFORM(SendsPayment)

```
((SendsPayment
    (agent-identifier
      :name customer0@192.168.0.2:1099/JADE
      :addresses (sequence http://francesco-pc:7778/acc)) 31320.0 3))
```

C Source Code

The code snippets shown below have been simplified: some of the methods contained in the classes, such as `equals()`, `toString()` and `hashCode()` have been omitted for display purposes. Getters and setters have been compressed into pseudo code strings. Most of the code has been skipped and only the relevant parts have been kept.

C.1 Computer, Desktop and Laptop

Listing 2: Computer abstract class

```
public abstract class Computer implements Concept {
  private Ram ram;
  private HardDrive hardDrive;
  private Os os;
  private Cpu cpu;
  private Motherboard motherboard;

  @Slot (mandatory = true)
  public Ram getter and setter {} // pseudo code to simplify

  @Slot (mandatory = true)
  public HardDrive getter and setter {}

  @Slot (mandatory = true)
  public Os getter and setter {}

  @Slot (mandatory = true)
  public Motherboard getter and setter {}

  @Slot (mandatory = true)
  public Cpu getter and setter {}
}
```

Listing 3: Desktop concrete class

```
public class Desktop extends Computer {
  private CpuDesktop cpu;
  private MotherboardDesktop motherboard;

  // Constructor
  public Desktop() {
   this.cpu = new CpuDesktop();
   this.motherboard = new MotherboardDesktop();
  }
```

```
  @Slot(mandatory = true)
  public CpuDesktop getter and setter {}

  @Slot(mandatory = true)
  public MotherboardDesktop getter and setter {}
}
```

Listing 4: Laptop concrete class

```
public class Laptop extends Computer {
  private CpuLaptop cpu;
  private MotherboardLaptop motherboard;
  private Screen screen;

  // Constructor
  public Laptop() {
   this.cpu = new CpuLaptop();
   this.motherboard = new MotherboardLaptop();
   this.screen = new Screen();
  }

  @Slot(mandatory = true)
  public CpuLaptop getter and setter {}

  @Slot(mandatory = true)
  public MotherboardLaptop getter and setter {}

  @Slot(mandatory = true)
  public Screen getter and setter {}
}
```

C.2 Component delivery times

Listing 5: Supplier sets the order delivery time

```
// When each of the suppliers is created, its prices and speed are addedd to the
    componentsForSale and suppDeliveryDays variables
Object[] args = getArguments();
if (args != null && args.length > 1) {
  componentsForSale = (HashMap<ComputerComponent, Integer>) args[0];
  suppDeliveryDays = (int) args[1];
}

...
```

```
// the supplier order wrapper object is used to hold data regarding a component order
public class SuppOrderWrapper {
  private AID buyer;
  private int deliveryDay;
  private ArrayList <ComputerComponent> components;
  private int quantity;

  public int getDeliveryDay() {
    return deliveryDay;
  }
  public void setDeliveryDay(int deliveryDay) {
    this.deliveryDay = deliveryDay;
  }

  // Other getters and setters omitted
}

...

// When an order is received, the supplier sets its delivery date. "day" indicates the
    day count
order.setDeliveryDay(day + suppDeliveryDays);
```

Listing 6: Supplier logic for delivery times

```
public class SendComponents extends OneShotBehaviour {
  public SendComponents(Agent a) {
    super(a);
  }

  @Override
  public void action() {
    for (SuppOrderWrapper order : orders) {
      // send all orders that have a delivery date equals to today's date
      if (order.getDeliveryDay() != day) continue;

      // Skipped code - send components
    }
  }
}
```

C.3 Warehouse storage cost

Listing 7: Warehouse storage cost calculation

```
// hashMap containing the components available to build computers and their quantity
private HashMap<ComputerComponent, Integer> warehouse = new HashMap<>();

...

// Here we are receiving the components from the supplier. shipComponents is the
    predicate used for this
ArrayList<ComputerComponent> compList = shipComponents.getComponents();
int quantity = shipComponents.getQuantity();

// These are added to the warehouse
for (ComputerComponent comp : compList) {
  if (warehouse.get(comp) == null) {
    warehouse.put(comp, quantity);
  } else {
    warehouse.put(comp, warehouse.get(comp) + quantity);
  }
}

...

// At the end of the day, calculate the storage fee
for (ComputerComponent comp : warehouse.keySet()) {
  double loss = warehouse.get(comp) * 5;
  dailyProfit -= loss;
}
```

C.4 Component availability

Listing 8: Component availability calculation

```
public class ManufactureAndSend extends OneShotBehaviour {
...
// This behaviour executes once a day
// for each order to manufacture, check that there are enough components
for (OrderWrapper orderWpr : orders) {

  // if the components for this order were not received, continue to the next order
  if (orderWpr.getOrderState() != OrderWrapper.State.COMPS_RECEIVED) continue;

  Boolean allCompsAvailable = true;
  // getComponentList() returns an ArrayList with all the components in an order
  for (ComputerComponent comp : orderWpr.getOrder().getComputer().getComponentList()) {
    if(!warehouse.containsKey(comp) ||
        (warehouse.containsKey(comp) && warehouse.get(comp) <
           orderWpr.getOrder().getQuantity())) {
```

```
      // if the warehouse does not contain a component, or if it does not contain a
         high enough quantity, set to false and stop the loop
      allCompsAvailable = false;
      break;
    }
  }

  // if not all components are available, don't proceed with the shipment
  if (!allCompsAvailable) continue;

  // Skipped code that sends the order and changes its state to AWAITING PAYMENT

  // Remove the used components from the warehouse
  for (ComputerComponent comp : orderWpr.getOrder().getComputer().getComponentList()) {
    warehouse.put(comp, warehouse.get(comp) - orderWpr.getOrder().getQuantity());
  }
}
}
```

C.5 Late delivery fee

Listing 9: Manufacturer order wrapper class

```
// The class ObjectWrapper was created to provide the manufacturer with order meta
    data and utility methods.
public class OrderWrapper {
  ...
  private Order order;
  private int orderedDate;

  public Order Order getter and setter {}; // pseudo code to simplify
  public int OrderedDate getter and setter { }

  // Return the exact date that an order needs to be delivered by
  public int getExactDayDue() {
    return this.order.getDueInDays() + orderedDate;
  }
}


// At the end of the day, the manufacturer calculates the late orders fees
public class EndDay extends OneShotBehaviour {
...
  // In action
  for (OrderWrapper orderWpr : orders) {
    // Calculate and subtract penalty for each late order. "day" holds today's date
```

```
    if (orderWpr.getExactDayDue() <= day) {
      lateDelivPenalty += 50;
    }
  }
}
```

C.6 Profit calculation

Listing 10: Profit calculation logic

```
// the manufacturer holds variables to track the daily profit
private double ordersShipped = 0, lateDelivPenalty = 0, warehouseStorage = 0,
    suppliesPurchased = 0, totalProfit = 0;

...

// once orders are shipped, customers send a payment to the manufacturer, which
    updates the variable ordersShipped
public class ReceivePayment extends Behaviour {
  ...
  if (ce instanceof SendsPayment) {
    SendsPayment sendPayment = (SendsPayment) ce;

    // locate which order this payment is for
    OrderWrapper orderWrp = orders.stream()
      .filter(o -> o.getOrder().getOrderId() == sendPayment.getOrderId())
      .findFirst().orElse(null);
    // flag the order as paid
    orderWrp.setOrderState(OrderWrapper.State.PAID);

    // Add received amount to daily profit
    ordersShipped += sendPayment.getMoney();
  }
}

...

// When component orders are made, the variable suppliesPurchased is updated. The
    following is part of a multi-step behaviour
case 3:
  // The order's total cost was previously calculated for this supplier
  SendsPayment sendPayment = new SendsPayment();
  sendPayment.setBuyer(myAgent.getAID());
  sendPayment.setMoney(orderWpr.getTotalCost());

  try {
```

```
    getContentManager().fillContent(payMsg, sendPayment);
    send(payMsg);

    // add the cost that will finally be subtracted to the profit
    suppliesPurchased += orderWpr.getTotalCost();

    step = 0;
  } catch (CodecException ce) {
    ce.printStackTrace();
  } catch (OntologyException oe) {
    oe.printStackTrace();
  }
break;

...

// the lateDelivPenalty and warehouseStorage are calculated at the end of the day. The
    following is contained in the EndDay one-shot behaviour
// Calculate and subtract penalty for late delivery for each order
for (OrderWrapper orderWpr : orders) {
    if (orderWpr.getExactDayDue() <= day) {
      lateDelivPenalty += 50;
    }
  }

// Calc storage fee
for (ComputerComponent comp : warehouse.keySet()) {
  double loss = warehouse.get(comp) * 5;
  warehouseStorage += loss;
}


// Finally, calculate the total profit
totalProfit += ordersShipped
    - lateDelivPenalty
    - warehouseStorage
    - suppliesPurchased;


// Subsequently, reset the variables used to update the total profit
```