MULTI-AGENT SYSTEM SUPPLY CHAIN COURSEWORK REPORT

Introduction

The smartphone supply chain problem is an instance of the supply chain problem and refers to the bidding, production, sourcing of supplies and delivery of smartphones in a supply chain. The supply chain problem itself is the problem of the planning and coordination of these activities with the ultimate goal of maximising profit. In a modern, fast-evolving world the supply chain problem remains an important and relevant issue in the building of electronic marketplaces and improvement of supply chain performance. (Arunachalam & Sadeh, 2005)

In recent times the need for faster, more efficient and adaptive systems has increased. As time-based competition has become more intense with the rise of manufacturers and marketplaces, and product design cycles shorten there is a greater need for collaborative production between designers, distributers, suppliers or market researchers. (Lee & Kim, 2008) A well-designed cooperative system can optimise the design process, planning and scheduling, and can be adaptive in unexpected situations. Agent-based systems allow the entire supply chain to be represented as a networked system of independent agents, each of which utilises its own decision making procedures (Gjerdrum, Shah, & Papageorgiou, 2001).

Multi-Agent systems are particularly suited to supply chain problems because they are able to apply autonomy, responsiveness, redundancy and openness to the system (Monostori, Váncza, & Kumara, 2006). Furthermore, as these agent-based systems are designed to handle dynamic systems through adaptivity, they are able to evaluate a much larger number of options than a human could, resulting in a significant increase in supply chain trading performance (Arunachalam & Sadeh, 2005).

The problem of the smartphone supply chain is as important as ever at the peak of smartphone production and can easily profit from the use of multi-agent systems to control and manage the supply chain. While complex multi-agent systems are well understood and applied to many supply chains, as the world ever-advances, it is clear that research has yet to fully exploit the potential of these systems (Lee & Kim, 2008).

Model Design

I have designed a complex multi-agent system capable of facilitating the ordering, supplying, building and delivery stages in a smartphone supply chain. Included in the supply chain are agents representing the customers, the manufacturer and the suppliers. The tickerAgent is not included in this report as its responsibility is synchronising the agent actions by controlling the structure of days and is not an agent directly involved in the supply chain. I have also chosen not to include the agent-finding classes and registration with the yellow pages, as these are classes not directly involved with the supply chain.

The customer agents are responsible for generating phone orders and sending each order proposal to the manufacturer. Each customer must generate one random order each day and record which orders have been accepted by the manufacturer. Each customer must also listen for deliveries every day.

The manufacturer agent is the largest agent in the system and is responsible for communicating with the customer and supplier agents, receiving and placing orders, building phones and delivering phones. Like any manufacturer, the main goal of this agent is to make finished products from raw materials. In this case the manufacturer must build phones from components acquired from the suppliers. The manufacturer must

communicate with both customer and supplier agents adhering to FIPA specifications, and use a structured process in order to build phones and maximise profits.

The supplier agents are each responsible for a stock of components which they must sell for a set price. Each supplier must communicate with the manufacturer in order to facilitate the successful sale of each component type.

The ontology featured in my system uses a combination of concept, agentAction and predicate classes in order to provide functionality to the system and allow agents to communicate with each other. In this section I will justify the design of each ontology class making reference to Appendix 1.

The ontology features four concepts representing each type of phone component: Screen, Storage, RAM and Battery. Each of these classes represents the appropriate component type and contains a single mandatory value representing the size of the component. I chose to include each of these components as concepts as they are all entities that 'exist' and that agents talk about, trade and store. Each of these concepts represents a different component, rather than *a* component because a phone must contain one of each type and this is made clear when each type is a separate concept.

In order to facilitate the sale of components I have designed an agentAction- 'sell'. The sell action represents a sale that is executed by the supplier agents. The sell class contains information about the component that is for sale, including the quantity, price and due date of the component delivery. In order to represent the component itself, rather than include fields for each component type, I include a 'component' class in order to simplify the selling process and message passing between agents.

The component class inherits the name and size from any of the four components. Using this class, I can include a component of any type in the sell action, and easily receive and decode the messages in each agent because there is a single field of type component rather than four separate classes. Including all four component classes would result in many non-mandatory fields and could be more confusing when trying to construct messages.

The primary concept featured in my ontology is the phone class. A phone contains a type, either 'small' or 'phablet' and one of each component. Every field in the phone class is compulsory- the phone must be of type small or phablet, and must contain a screen, storage, RAM and battery. Without any of these fields the phone would not be a complete representation of a phone, and therefore not a phone. I include this concept to represent a full phone. I decided against including separate classes for small phones and phablets because each of these types can still be defined as a phone, regardless of the size, and each phone can be defined as one type or the other in the phone class.

The other agent actions included in my ontology are 'orderInst' and 'deliver'. The orderInst action is used by the customer to order phones from the manufacturer. Using this as an action the customer is able to order the quantity of phones they require. This action also includes all the mandatory information included in an order including the unit price and late penalty. The manufacturer uses the deliver action to deliver phones to the customer. This action contains the specification of the phone, the quantity of phones and the cost of the order.

Both the orderInst and delivery actions are essential to my ontology as they represent an action, rather than physical objects which exist. These actions facilitate communication about the objects and the transfer of ownership of the objects.

The owns predicate class is used to signal ownership of a component by the supplier. I included this predicate class so that the manufacturer can ask if the supplier *owns* the component.

Referring to Appendix 2, Figure 1, the communication protocols I have used in the initial conversation between the customer and manufacturer are appropriate for sending an order and awaiting a response. The protocol I have used for sending the offer to the manufacturer is a request. This is because the customer is requesting that the manufacturer perform the action of building their order. The manufacturer then replies to the request with an AGREE or a REFUSE, stating whether they agree to build the order or they refuse to build it. I avoided using a propose as this conversation is a request rather than a negotiation.

The conversation between the manufacturer and supplier seen in Figure 2 uses a range of negotiation communication protocols. The manufacturer first uses a query-if message in order to request information- whether the component is in stock. If true, this follows with a CFP, in which the supplier is calling for the manufacturer to propose. I have included the call for proposal, and the following proposal from the manufacturer because these performatives are negotiation performatives and the sale of a component is a negotiation between the manufacturer and the supplier.

The final conversation seen in Figure 3 between the manufacturer and the customer uses an inform type message to inform the customer that an order is being delivered. I used this communication protocol because it is an appropriate message type for passing information from one agent to another, in this case a delivery and its contents.

Model Implementation

Detailed in this section are descriptions of the behaviours each agent executes and their behaviour type, the communication protocols and a description of how the implementation meets the constraints of the task listed in Section 3 of the task specification. The code referenced in this section is included in Appendix 3.

The ‘tickerWaiter’ of type cyclic behaviour is the first behaviour to appear in every agent. The purpose of this behaviour is to listen out for a message from the tickerAgent signalling a new day is beginning. At the start of each new day the agents can begin executing their behaviours. The tickerWaiter behaviour is able to do this because it contains a sequential behaviour- ‘dailyActivity’, which executes its children behaviours one by one before terminating. These children behaviours are the behaviours executed by each agent every day. An example of the use of this behaviour in the manufacturer agent can be seen in Figure 1.

The ‘orderListener’ behaviour waits to receive a message from every customer agent before it finishes. This behaviour receives the requests, as seen in Figure 2, and determines which order it would like to accept by extracting the ontology from the message and calculating the potential profit of each order. Once the behaviour has selected the highest profit, an agreement is sent to the customer with the best order, and refusals are sent to the remaining customers. This process can be seen in Figure 3.

After the selecting an order, the ‘orderParts’ behaviour is called. This behaviour is one-shot because the manufacturer only orders parts once per day. If an order is not accepted on a given day, no parts will be ordered. The parts are ordered from one or both suppliers depending on the due date of the order to ensure parts are delivered in time to avoid late fees. Figure 4 shows the conversation for one component. For each component the behaviour starts by querying if the component is in stock. This query-if message sends the

name and size of the component wrapped in an owns predicate. The behaviour then waits for a response and can handle either a CFP or REFUSE response. If the item is confirmed to be in stock, the behaviour sends a proposal, confirmation of the intent to buy the given quantity of component. The negotiations and sale of all 4 components are completed in this one shot behaviour using the owns and sell predicate and action classes from the ontology.

The customer agent begins by executing the 'queryBuyerBehaviour' one-shot behaviour in which it generates an order and sends a message of type propose containing the order for that day. These can be seen in Figure 5. This behaviour is of type one-shot, as it only needs to be executed once per day. The message is of type request because the customer is requesting that the manufacture agent perform an action (building the phone).

The customer agent also contains a behaviour to receive an order- 'ReceiveDeliveryListener'. This behaviour is a cyclic behaviour and is required to listen constantly for a delivery of phones. The listener matches messages with performative INFORM and conversation ID 'delivery' and represents the completion of an order. Each delivered order is removed from the customers list of open orders. This process can be seen in Figure 6.

The first behaviour called by the supplier agents is 'queryBehaviour'. This behaviour is responsible for receiving and sending messages regarding the supply of components to the manufacturer, as seen in Figure 7. The behaviour begins by listening for a message with query-if performative, and confirms whether it *owns* the component or not. The behaviour continues by listening for a message containing the sell action and responds by sending a 'confirm' message, completing the sale of the component. This behaviour listens for four messages- one for each component. This can be seen in Figure 8.

Both the 'supplier1' and 'supplier2' agents contain the same behaviours. However, because a maximum of 2 components are ordered from supplier2 each day, the behaviour detailed above is also tuned to listen for an inform message signalling that the manufacturer has finished ordering. This can be seen in the first few lines of Figure 7. This ensures that the supplier agents are able to communicate with the tickerAgent that their daily activities have been completed and that they are ready for a new day.

In demonstrating satisfaction of the constraints, I detail below how and where each constraint has been successfully implemented.

Constraint 1, referring to 'small' and 'phablet' phones and the components in each is satisfied in the generation of phone orders in the customer class. Each phone is assigned a type based on a random number, and cannot be assigned the wrong screen or battery based on its type. This can be seen in Figure 5.

Constraint 2, referring to the component delivery times is satisfied in Figure 8, where the supplier 1 applies the delivery time in days. This process is the same for supplier 2 but with a different due date. The manufacturer adds the order to a list as seen in Figure 9. Each day the manufacturer checks the delivery date of each item in transit to check if it is being delivered that day or not, as seen in Figure 10.

Constraints 3 and 7 are satisfied in the expenditure class of the manufacturer. It is in this class that the storage costs are calculated and the correct calculation of profit happens each day. This can be seen in Figure 11.

Constraint 4, stating that orders can only be shipped if there are sufficient components in stock is satisfied in Figure 12, in which the behaviour loops through the orders in due-date-

ascending order checking if each order can be built. In a second behaviour designed to try and build another order based on the number of smartphones built already constraint 5 is satisfied. This can be seen in Figure 13.

Finally, the penalties for late deliveries, constraint 6, are applied in Figure 14, where late fees are calculated before any orders are built each day.

Design of Manufacturer Agent Control Strategy

The manufacturer agent decides which order to accept based on the potential profit of each order. Once every order has been received from the customer agents, the potential profit from each order is calculated by multiplying the price per unit by the quantity of smartphones ordered. Using this strategy, the offer with the highest potential profit is always selected.

In a system with 3 customers, selecting the lowest and middle offers, or a mixture of both, often resulted in the manufacturer making a loss at the end of 100 days. Additionally, basing the selection on other variables such as the due date and per-day penalty resulted in lower profit margins. Therefore I justified using this selection process in order to maximise profit in the system.

Always accepting the order with the highest potential profit it is very likely that only one order can be constructed and delivered each day. By day 90 this usually results in a backlog of several orders. In order to ensure that each and every order has been built, in days 90-100 the manufacturer changes the selection strategy and will only accept orders in which the due date is before or on day 100. This usually results in around 7 orders being accepted, leaving 3 days for orders that have been delayed to be built and delivered.

Using this system results in a fairly high probability of an order being accepted in days 90-98. This allows each day that no order is accepted to be used for building orders currently in the open orders list to be constructed. This strategy is extremely useful for ensuring that there is not a backlog of incomplete orders at the end of the simulation. Implementing this feature also saves several thousand pounds in the final days by avoiding buying parts from suppliers for orders that are not accepted because they will not be delivered in time.

In order to select which supplier agent to order parts from, the manufacturer uses the due date of the order to select the most cost-effective supplier. In orders where the due date is 3 days or under, the manufacturer will always order all of the parts from supplier 1 who delivers on the next day. If the due date is 4 days – 10 days, the manufacturer will order the screen and battery components from supplier 1, as this is the only supplier that stocks these components, and the storage and RAM from supplier 2 for delivery on day 4 from the order being placed. The amount of components ordered is always equal to the quantity of phones in the accepted order.

Components are ordered in my system in a buy-as-needed system, where there is no incentive to buy components that are not needed for an existing order. This is also an ideal strategy for trying to keep storage costs as low as possible. The idea of ordering components early to form a stock was inspired by the supply chain trading competition (Arunachalam & Sadeh, 2005), in which components are ordered early at a reduced price. However due to the different structure to this supply chain, I determined there is no incentive to purchase components earlier than needed.

Using this control strategy when ordering parts allows the manufacturer to save as much money as possible on components by allowing the components for orders with a higher due date to be delivered later at a smaller overall cost. Over the course of the 100 days this

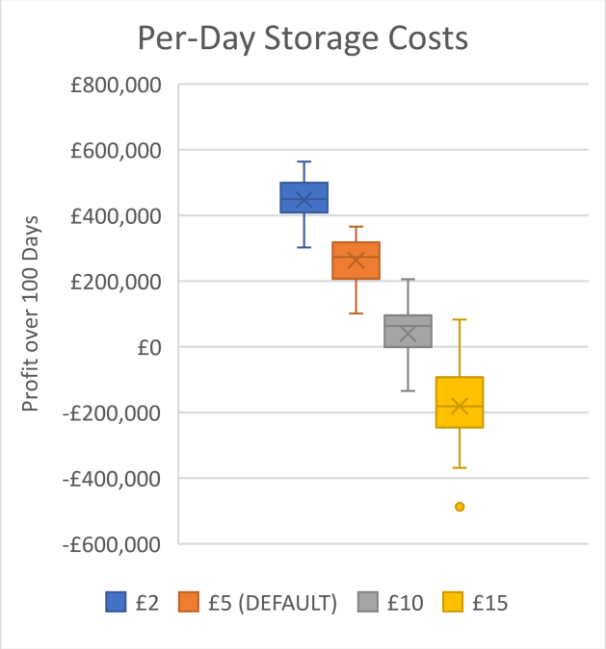
reduces the amount of money spent on components, and helps to maximise profit. Components on delivery are always in theory placed in the warehouse, however for the purpose of this system I assume that only components present in the warehouse at the end of each day have been placed in storage, and storage fees will only be applied to these components.

In choosing which order to deliver goods, the manufacturer sorts the list, with the lowest due date (compared to the day) always being selected first. The report on the supply chain trading competition (Arunachalam & Sadeh, 2005) discusses the process of deciding based on both the due date and the late penalties, and although my system only considers the due date when selecting which order to build, the purpose of this is to reduce the late fees acquired. A future version may incorporate the late fee in selecting which late order is prioritised.

Experimental Results

Each experiment featured in this section consists of data collected over the course of 40 full runs of the system and will be presented alongside the exact changes made to the system. The default parameters for the system are as follows: 3 customers, £5 per day per component storage fee, late fee determined by quantity*(1+50*random), always select the order with the highest potential profit, use supplier 1 and 2.

I will be testing my system from 3 different approaches in order to test its robustness and ability to deal with dynamic inputs. The first experiments I perform will test how the system performs with increased daily costs. This includes costs relating to component storage and order late fees. My system should be robust and able to handle changes to the daily costs associated with the storage and late fees. Depending on the system's ability to build orders quickly and avoid late fees, in changing these parameters I would expect to reduce the profit.



The second tests I am performing on my system involve the order selection. It is unlikely that changing either the order accepted or use of suppliers will result in higher profits. I have decided not to test the process of ordering components as I have previously determined that using both suppliers is advantageous over using only supplier 1 in reducing overall component costs.

The third test compares a system with default parameters with parameters set to maximise profit. This experiment shows how the system would work in an ideal situation with low daily costs and a large range of orders to choose from per day. I expect this experiment to result in very high profits for one system.

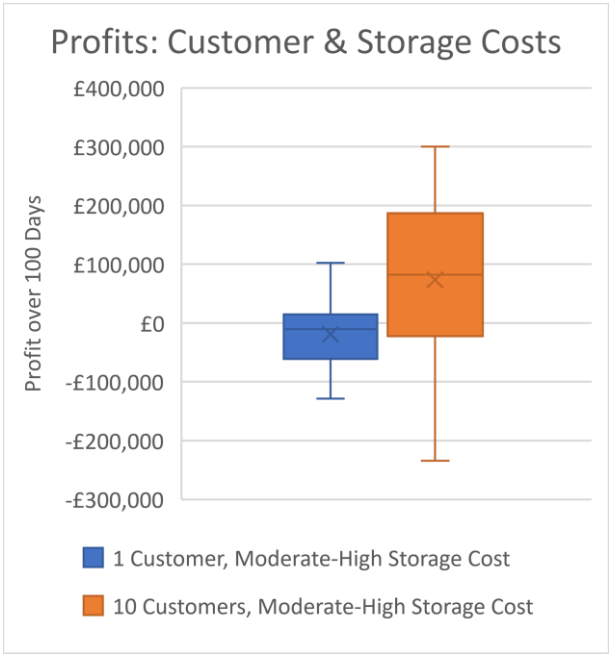
In testing how the system performs with higher daily costs, I performed two experiments- the first dealing with increased storage costs and the second with increased late fees. In this experiment I alter the storage cost to £2, £10 and £15 in order to test how the system handles different prices. As expected, a lower cost of £2 per day results in a significant increase in profit. Increasing the component cost to £10 reduces the profit significantly, however in most cases the system remains profitable, with an average profit of £39,676. Increasing the storage cost further to £15 results in a system which is rarely profitable.

Accepting the order with the highest profit each day is likely responsible for this due to the much larger quantity of phones ordered on average. The results can be seen clearly in the ‘per-day storage costs’ box and whisker diagram.

Increasing the late fees produces a similar trend of reduced profits with higher costs, however due to the effectiveness of the system at avoiding late fees, even with doubled late fees the system returns an average profit of £174,829. Reducing the late fees had a very small effect on the profit, with halved late fees only increasing the system profit by 13.52%.

In the second stage of experiments I change the selection process in order to see which combination of orders results in the highest profits. Interestingly selecting the largest offer every time is significantly more profitable than selecting the middle or lowest offers and produces the largest range of profit margins. The difference between the middle offer and the highest offers is huge, with the middle offers only once making profit above £100,000. Accepting a combination of medium and largest orders would likely result in a profitable system every time, however this would not result in higher profits than selecting the largest every time.

In order to fully test the effect of increased storage costs, and to reflect a real system in which the amount of customers varies, I used moderate-high storage costs (£10) and compared how the system performs with 1 customer and 10 customers. Perhaps the most interesting of results, I found that with a higher range of offers per day, the system is much more volatile with a standard error of £25,667. This likely relates to the amount of outstanding orders the system carries with it through the process. As smaller orders usually have smaller quantities, in a situation with 1 customer the system can build 2 orders per day more often and therefore reduce the amount of parts kept in storage. However, with 10 orders each day and always accepting the biggest, the warehouse is likely to store more components through the system due to the reduced probability of building more than one order and the increased probability of large quantities of phones being ordered each day.



In the final stages of experimentation I compared the default system against a system with parameters known to increase profit in order to discover how much higher the profit can be using better parameters. In the new system I decrease late fees to 1/5 the original cost, decrease storage costs to £2 per component per day and increase the customer count to 10 in order to increase the chances of receiving extremely profitable orders. Using this combination of parameters results in a system with

consistently very high profits.

	Average	Std. Error
Default Parameters	£262,610	£12,844
Max Parameters	£736,813	£12,554

Conclusions

The design of my multi-agent system is particularly suited to a simplistic supply chain with one type of product, simple manufacturing with no building costs, instant delivery of goods to the customer, and no possibilities of miscommunication, order cancelling and errors in the manufacturing line. This supply chain represents a basic version of a real supply chain with all unexpected events removed. In a more realistic version of the supply chain the multi-agent system must be able to handle unexpected situations in a responsive and adaptive way.

The biggest way in which my system could be expanded to handle a more realistic supply chain scenario is the addition of processes to handle order cancellation or modification of an accepted order. In a real supply chain orders are frequently modified after they have been placed whether the order is cancelled or the quantities are increased or decreased. A customer might wish to change the storage specifications of the phones they have ordered or downgrade the RAM. In my current system there is no facilities for changing orders once they have been placed.

The system could be expanded to handle a more realistic supply chain by introducing a more realistic banking system. Currently the manufacturer is able to order components on the first day with no money in the bank, allowing it to immediately go into debt. Introducing a banking system with real overdraft fees and bills, machinery costs and other expenditures would be more comparable to a real supply chain.

Another way in which the system could be modified is allowing the supplier agents to have control over the cost of components. The component cost is likely to fluctuate over time based on supply and demand and changing material prices, as seen in the supply chain trading agent competition (Arunachalam & Sadeh, 2005). This would in turn likely affect elements of the manufacturing process.

The manufacture control strategy already has measures in place to reduce the likeliness of accepting orders it cannot complete before the 100 days is up, and also tries to ensure every accepted order is constructed by the end of this period. In improving this strategy though, profit could be increased if orders accepted in the final 10 days are based on potential profit and due date, rather than just due date. This would allow the manufacturer to accept the orders with the highest profit in the final days that can be constructed in time for the end.

Another way in which profit could be maximised in my system is allowing two orders to be accepted on a given day if they had a higher combined profit than just the highest single order. On occasion this might mean accepting the smallest and medium offer, or the highest and the smallest. The only constraint that this technique would need to satisfy is that no more than 50 phones are accepted in a single day.

A further improvement would be to try and predict the likeliness of an order being delivered late based on the current stock and manufacturing situation, and perhaps accept orders with less likeliness of being delivered late. This would avoid situations in which multiple orders are all due around the same date, and ultimately avoid paying as many late fees.

References

Arunachalam, R., & Sadeh, N. M. (2005). THE SUPPLY CHAIN TRADING AGENT COMPETITION Raghu Arunachalam , Norman M . Sadeh CMU-CS-04-164 School of Computer Science Carnegie Mellon University Pittsburgh , PA 15213-3891 Also appears as Institute for Software Research International Technical Report C. *Electronic Commerce Research & Applications Journal*.

Gjerdrum, J., Shah, N., & Papageorgiou, L. G. (2001). A combined optimization and agent-based approach to supply chain modelling and performance assessment. *Production Planning and Control*, *12*(1), 81–88. https://doi.org/10.1080/09537280150204013

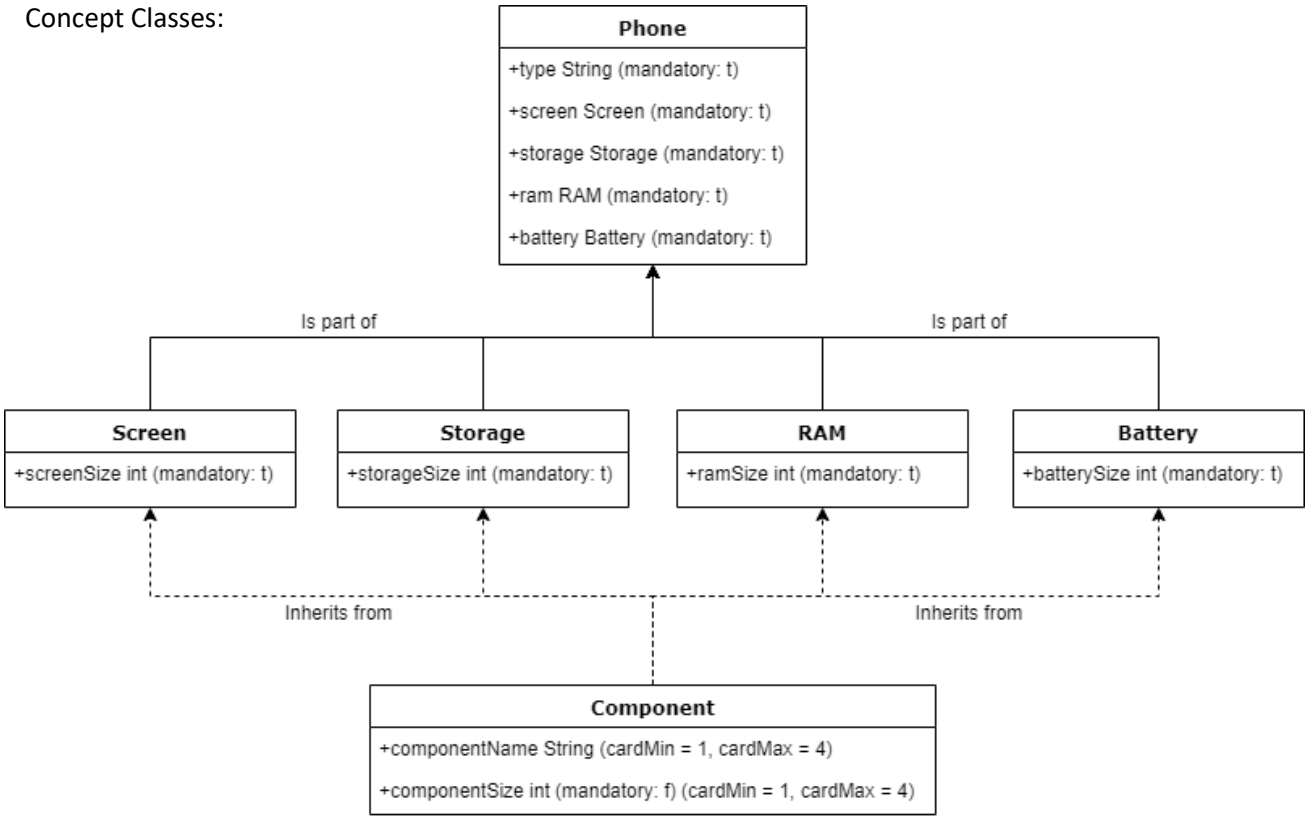
Grant, I. T. R., & Chain, S. (2004). THE SUPPLY CHAIN TRADING AGENT COMPETITION. *Science*.

Lee, J. H., & Kim, C. O. (2008). Multi-agent systems applications in manufacturing systems and supply chain management: A review paper. *International Journal of Production Research*, *46*(1), 233–265. https://doi.org/10.1080/00207540701441921

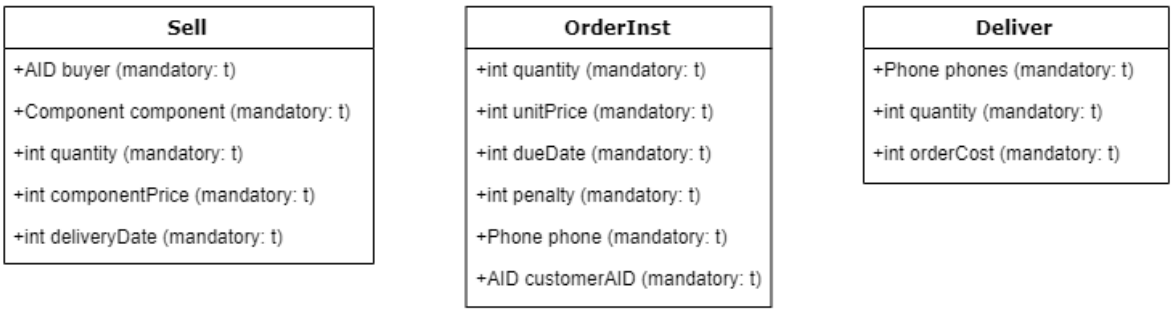
Monostori, L., Váncza, J., & Kumara, S. R. T. (2006). Agent-based systems for manufacturing. *CIRP Annals - Manufacturing Technology*, *55*(2), 697–720. https://doi.org/10.1016/j.cirp.2006.10.004

Appendix 1: Ontology

Concept Classes:



AgentAction Classes:



Predicate Classes:

Examples:

Screen

screenSize = 7

Storage

storageSize = 256

RAM

ramSize = 8

Battery

batterySize = 3000

Phone

type = “phablet”

screen = ontology.elements.Screen@564145c7

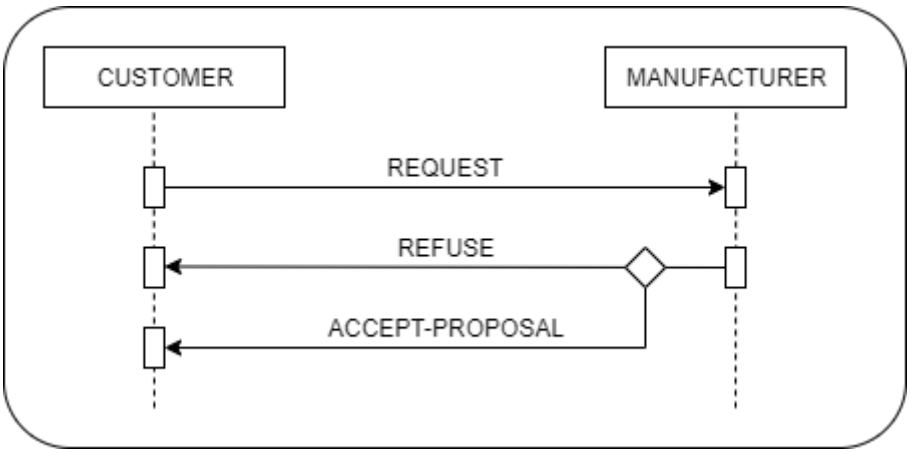
storage = ontology.elements.Storage@4da3c4eb

ram = ontology.elements.RAM@7ld3d8ab

battery = ontology.elements.Battery@542673e4

Appendix 2: Communication Protocols

Figure 1 – Communication Between Customer and Manufacturer



REQUEST

```
((action
    (agent-identifier
     :name Manufacturer@192.168.0.40:1099/JADE)
    (OrderInst
     :customer
      (agent-identifier
       :name Customer2@192.168.0.40:1099/JADE
       :addresses (sequence http://DESKTOP-
N5EQ5V2:7778/acc))
     :dueDate 2
     :penalty 1353
     :phone
      (Phone
       :battery
        (Battery
         :componentName Battery
         :componentSize 0
         :batterySize 3000)
       :rAM
        (RAM
         :componentName RAM
         :componentSize 0
         :ramSize 4)
       :screen
        (Screen
         :componentName Screen
         :componentSize 0
         :screenSize 7)
       :storage
        (Storage
         :componentName Storage
         :componentSize 0
         :storageSize 64))
     :quantity 33
     :unitPrice 184)))
```

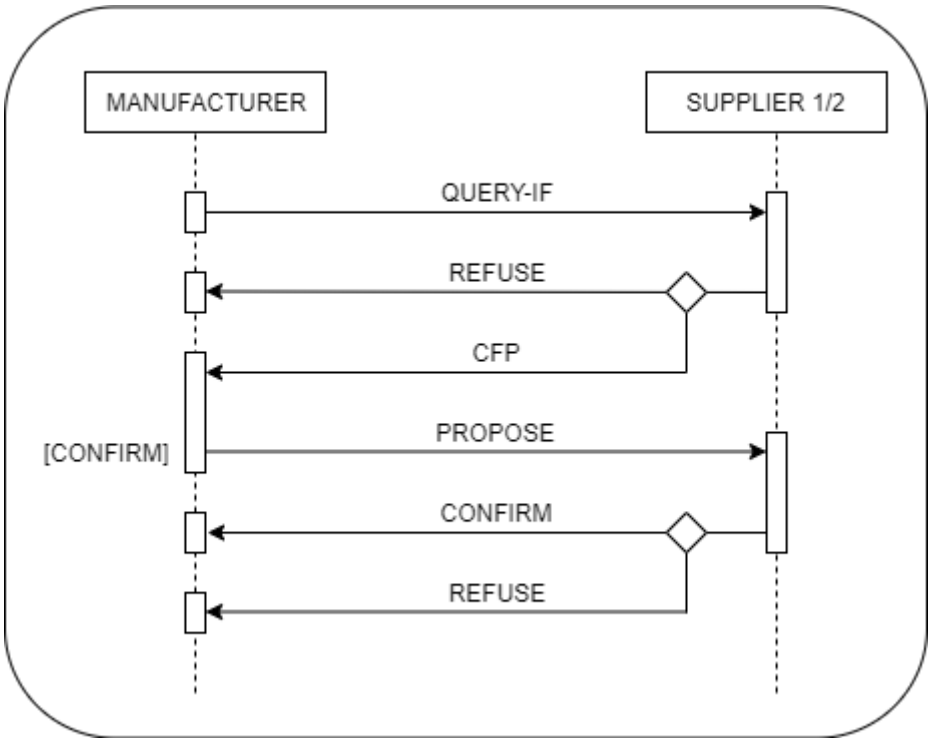
ACCEPT-PROPOSAL

Order Accepted.

REFUSE

Order Refused.

Figure 2 – Communication Between Manufacturer and Suppliers 1 & 2



QUERY-IF

((Owns

(Component

:componentName Screen

:componentSize 5)

(agent-identifier

:name Supplier1@192.168.0.40:1099/JADE) 41))

REFUSE

Component not in stock.

CALL FOR PROPOSAL

((Owns

(Component

:componentName Screen

:componentSize 5)

(agent-identifier

:name Supplier1@192.168.0.40:1099/JADE) 41))

PROPOSE

((action

(agent-identifier

:name Supplier2@192.168.0.40:1099/JADE

:addresses (sequence http://DESKTOP-N5EQ5V2:7778/acc))

(Sell

:buyer

(agent-identifier

:name Supplier2@192.168.0.40:1099/JADE

:addresses (sequence http://DESKTOP-N5EQ5V2:7778/acc))

:component

(Component

:componentName Storage

:componentSize 256)

:componentPrice 0

:quantity 41)))

CONFIRM

((action

(agent-identifier

:name Manufacturer@192.168.0.40:1099/JADE

:addresses (sequence http://DESKTOP-N5EQ5V2:7778/acc))

(Sell

:buyer

(agent-identifier

:name Manufacturer@192.168.0.40:1099/JADE

:addresses (sequence http://DESKTOP-N5EQ5V2:7778/acc))

:component

(Component

:componentName Storage

:componentSize 256)

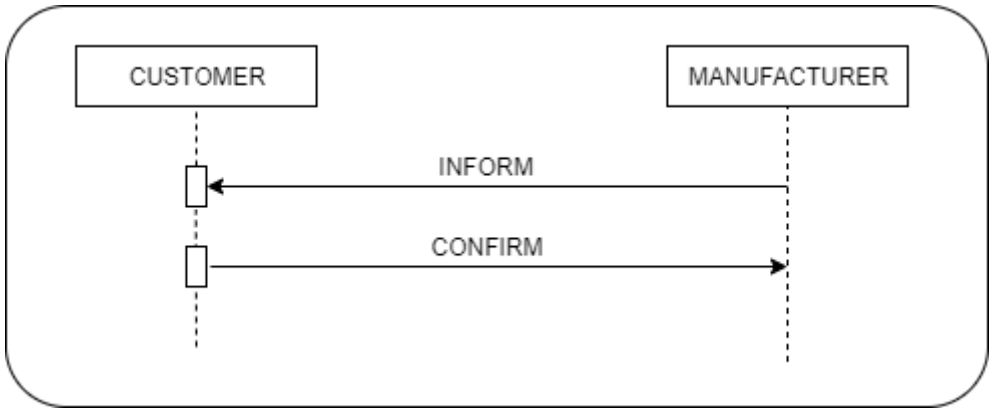
:componentPrice 40

:quantity 41)))

REFUSE

Sell refused.

Figure 3 – Communication Between Manufacturer and Customer on Delivery of Goods



INFORM

((action

(agent-identifier

:name Customer1@172.28.60.225:1099/JADE

:addresses

(sequence http://me0mc10-150602.napier-mail.napier.ac.uk:7778/acc))

(Deliver

:orderCost 9044

:quantity 28

:phone

(Phone
:battery
(Battery
:componentName Battery
:componentSize 0
:batterySize 3000)
:rAM
(RAM
:componentName RAM
:componentSize 0
:ramSize 4)
:screen
(Screen
:componentName Screen
:componentSize 0
:screenSize 7)
:storage
(Storage
:componentName Storage
:componentSize 0
:storageSize 64))
)))

CONFIRM

Delivery Received.

Appendix 3: Source Code

Figure 1

```
public class TickerWaiter extends CyclicBehaviour {
		// behaviour to wait for a new day
		public TickerWaiter(Agent a) {
			super(a);
		}

		@Override
		public void action() {
			MessageTemplate mt = MessageTemplate.or(MessageTemplate.MatchContent("new day"),
					MessageTemplate.MatchContent("terminate"));
			ACLMessage msg = myAgent.receive(mt);
			if (msg != null) {
				if (tickerAgent == null) {
					tickerAgent = msg.getSender();
				}
				if (msg.getContent().equals("new day")) {

					// Reset these values on each new day.
					noOrder = false;
					todaysCost = 0;
					income = 0;
					income02 = 0;
					lateFees = 0;

					// spawn new sequential behaviour for day's activities
					SequentialBehaviour dailyActivity = new SequentialBehaviour();
					// sub-behaviours will execute in the order they are added
					dailyActivity.addSubBehaviour(new OrderListener(myAgent));
					dailyActivity.addSubBehaviour(new orderParts(myAgent));
					dailyActivity.addSubBehaviour(new checkDeliveries(myAgent));
					dailyActivity.addSubBehaviour(new buildOrder1(myAgent));
					dailyActivity.addSubBehaviour(new buildOrder2(myAgent));
					dailyActivity.addSubBehaviour(new Expenditure(myAgent));
					dailyActivity.addSubBehaviour(new EndDay(myAgent));
					myAgent.addBehaviour(dailyActivity);
					day++;
				} else {
					// termination message to end simulation
					myAgent.doDelete();
				}
			} else {
				block();
			}
		}

	}
```

Figure 2

```
// Listen for all orders being placed.
public class OrderListener extends Behaviour {

	public OrderListener(Agent a) {
		super(a);
	}

	boolean finished = false;

	// Orders received from agents
	int messageReceived = 0;
	// The AID of the accepted order.
	AID accepted;
	// Step for switch case.
	private int step = 0;
	// Keeps track of the index of the received orders list.
	private int index = 0;
	// List of all received orders.
	ArrayList<OrderInst> orders = new ArrayList<OrderInst>();


@Override
public void action() {
	switch(step) {
	case 0:
		while(messageReceived != customerCount) {

			MessageTemplate mt = MessageTemplate.and(MessageTemplate.MatchLanguage(codec.getName()),
					MessageTemplate.MatchOntology(ontology.getName()));
			ACLMessage msg = blockingReceive(mt);

		if (msg != null) {
			try {
				ContentElement ce = null;
				if (msg.getPerformative() == ACLMessage.REQUEST) {
					ce = getContentManager().extractContent(msg);
					if (ce instanceof Action) {
						Concept action = ((Action)ce).getAction();
						if(action instanceof OrderInst)
						{

							// Extract and print order information
							OrderInst phoneOrder = (OrderInst)action;
							System.out.println("ORDER NO. " + (messageReceived + 1));
							System.out.println("Screen: "
							+ phoneOrder.getPhone().getScreen().getScreenSize() + "\""
							+ "    Storage: "
							+ phoneOrder.getPhone().getStorage().getStorageSize() + "Gb"
							+ "    RAM: "
							+ phoneOrder.getPhone().getRAM().getRamSize() + "Gb"
							+ "    Battery: "
							+ phoneOrder.getPhone().getBattery().getBatterySize() + "mAh"
							+ "    Quantity: " + phoneOrder.getQuantity() + " units"
							+ "    Unit Price: £" + phoneOrder.getUnitPrice() + " per unit"
							+ "    Due Date: " + phoneOrder.getDueDate() + " days"
							+ "    Penalty per day: £" + phoneOrder.getPenalty()
							+ "    AID: " + phoneOrder.getCustomer());
							System.out.println("");

							orders.add(phoneOrder);

							messageReceived++;
						}
						else {
							return;
						}
					}
				}
			}
			catch (CodecException ce) {
				ce.printStackTrace();
			} catch (OntologyException oe) {
				oe.printStackTrace();
			}
		}
	}
```

Figure 3

```
// Accept order based on highest potential profit.
// If the day is past 90 days, will only accept offers that have a due date of less than or equal to day 100.
// In order to give the manufacturer time to build any remaining orders left, past day 95, the manufacturer will only
// accept orders if there are less than 2 orders left to build. These orders must also have a due date of less
// than or equal to 100 days.

        ArrayList<Integer> maxIncome = new ArrayList<Integer>();
        int m = 0;

        while(m != customerCount) {
                maxIncome.add((orders.get(m).getQuantity() * orders.get(m).getUnitPrice()));
                m++;
        }

        int maxNo = Collections.max(maxIncome);

        index = maxIncome.indexOf(maxNo);

        if(day > 90) {

                int i = 0;
                int j = 0;
                int k = 0;

                if(day > 97 && openOrders.size() > 2) {

                        oS--;

                        while(j < customerCount) {
                                ACLMessage reply1 = new ACLMessage(ACLMessage.REFUSE);
                                reply1.addReceiver(customers.get(j));
                                myAgent.send(reply1);
                                j++;

                        }

                        noOrder = true;
                        System.out.println("Refusing all orders to catch up on order backlog.");
                        step++;
                        finished = true;
                        break;
                }


                while(k < customerCount) {
                        if((day + orders.get(i).getDueDate()) <= 100) {
                                openOrders.add(orders.get(i));
                                noOrder = false;

                                ACLMessage reply = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
                                reply.addReceiver(customers.get(i));
                                myAgent.send(reply);
                                totalOrdersTaken = totalOrdersTaken+1;
                                System.out.println("Manufacturer accepting order " + (i + 1) + ".");

                                k = customerCount;
                        }
                        else {
                                noOrder = true;
                                i++;
                                k++;
                        }
                }

                if(noOrder == false) {
                        while(j < customerCount) {
                                if(j != i) {
                                        ACLMessage reply1 = new ACLMessage(ACLMessage.REFUSE);
                                        reply1.addReceiver(customers.get(j));
                                        myAgent.send(reply1);
                                        j++;
                                }
                                else {
                                        j++;
                                }
                        }
                }


                if(noOrder == true) {

                        oS--;

                        while(j < customerCount) {
                                ACLMessage reply1 = new ACLMessage(ACLMessage.REFUSE);
                                reply1.addReceiver(customers.get(j));
                                myAgent.send(reply1);
                                j++;
```

```
                }
        }
        step++;
}
else {

        System.out.println("Manufacturer accepting order " + (index+1) + ".");
        openOrders.add(orders.get(index));
        accepted = orders.get(index).getCustomer();
        System.out.println("Accepting order from " + accepted);

        // ACCEPT ORDER ---------------------------------------------------------------------------------------------------------------
        if(noOrder == false) {
                ACLMessage reply = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
                reply.addReceiver(accepted);
                myAgent.send(reply);
                totalOrdersTaken = totalOrdersTaken+1;
        }

        step++;

        // DECLINE ORDERS -------------------------------------------------------------------------------------------------------------

        if(noOrder == false) {
                int z = 0;

                while(z != customerCount) {
                        if (z != index) {
                                ACLMessage reply1 = new ACLMessage(ACLMessage.REFUSE);
                                reply1.addReceiver(orders.get(z).getCustomer());
                                myAgent.send(reply1);
                                z++;
                        }
                        else {
                                z++;
                        }
                }
        }
        else if (noOrder == true) {

                int k = 0;
                while(k != customerCount) {
                        ACLMessage reply1 = new ACLMessage(ACLMessage.REFUSE);
                        reply1.addReceiver(orders.get(k).getCustomer());
                        myAgent.send(reply1);
                        k++;
                        System.out.println("Sending refuse to customer " + k);
                }
        }

        step++;

}
```

Figure 4

```
// Always order some parts from supplier 1. If the due date is 5+ days, set the supplierName variable
// to supplier 2. This means we will order the available parts from supplier 2 at a lower price.
// For each part, order, receive confirmation of the parts stock and send a sell request,
// and then receive confirmation of purchase. Move on to the next step.

        if(openOrders.get(oS).getDueDate() < 4) {
                supplierName = new AID("Supplier1", AID.ISLOCALNAME);
        }
        else if(openOrders.get(oS).getDueDate() >= 4) {
                supplierName = new AID("Supplier2", AID.ISLOCALNAME);
        }

        switch(step) {
        case 1:

                ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
                msg.addReceiver(supplier1);
                msg.setLanguage(codec.getName());
                msg.setOntology(ontology.getName());

                Component screenOrder = new Component();

                screenOrder.setComponentName("Screen");
                screenOrder.setComponentSize(openOrders.get(oS).getPhone().getScreen().getScreenSize());
                Owns screenRequest = new Owns();
                screenRequest.setOwner(supplier1);
                screenRequest.setComponent(screenOrder);
                screenRequest.setQuantity(openOrders.get(oS).getQuantity());

                try {
                        getContentManager().fillContent(msg, screenRequest);
                        send(msg);
                        step++;
                }
                catch (CodecException ce) {
                        ce.printStackTrace();
                }
                catch (OntologyException oe) {
                        oe.printStackTrace();
                }

        case 2:
                while(messageReceived != true) {
                        MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
                        ACLMessage msgR1 = receive(mt);
                        if(msgR1 != null){
                                try {
                                        ACLMessage response = new ACLMessage(ACLMessage.PROPOSE);
                                        response.addReceiver(msgR1.getSender());
                                        response.setLanguage(codec.getName());
                                        response.setOntology(ontology.getName());

                                        Component componentSale = new Component();

                                        componentSale.setComponentName("Screen");

        componentSale.setComponentSize(openOrders.get(oS).getPhone().getScreen().getScreenSize());
                                        Sell sellComp = new Sell();
                                        sellComp.setBuyer(msgR1.getSender());
                                        sellComp.setComponent(componentSale);
                                        sellComp.setDeliveryDate(day);
                                        sellComp.setComponentPrice(oS);
                                        sellComp.setQuantity(openOrders.get(oS).getQuantity());

                                        Action request = new Action();
                                        request.setAction(sellComp);
                                        request.setActor(msgR1.getSender());

                                        try {
                                                getContentManager().fillContent(response, request);
                                                send(response);
                                                messageReceived = true;
                                                //step++;

                                        }
                                        catch (CodecException ce2) {
                                                ce2.printStackTrace();
                                        }
                                        catch (OntologyException oe) {
                                                oe.printStackTrace();
                                        }
```

```
                        } catch (Exception e) {}
                    }
                }
                step++;

        case 3:

                int i = 0;
                while(i != 1) {

                        MessageTemplate sr = MessageTemplate.MatchPerformative(ACLMessage.CONFIRM);
                        ACLMessage msgS = receive(sr);
                        if(msgS != null){
                                try {
                                        ContentElement ceS = null;

                                        // Let JADE convert from String to Java objects
                                        // Output will be a ContentElement
                                        ceS = getContentManager().extractContent(msgS);
                                        if(ceS instanceof Action) {
                                                Concept action = ((Action)ceS).getAction();
                                                if (action instanceof Sell) {
                                                        Sell order = (Sell)action;
                                                        Component itS = order.getComponent();
                                                        Component compOrderS = (Component) itS;
                                                        order.setDeliveryDate((day + order.getDeliveryDate()));

                                                        todaysCost = (todaysCost + (order.getQuantity() *
order.getComponentPrice()));
                                                        openDeliveries.add(order);

                                                        i++;
                                                }
                                        }
                                } catch (Exception e) {}
                        }
                }
```

Figure 5

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

        boolean messageReceived = false;

        public void action() {
                int step = 0;
                OrderInst phoneOrder = new OrderInst();

                switch(step){
                case 0:

                        String type;
                        int screen;
                        int storage;
                        int ram;
                        int battery;
                        int quantity = (int) Math.round(Math.floor(1 + 50 * Math.random()));
                        int unitPrice = (int) Math.round(Math.floor(100 + 500 * Math.random()));
                        int dueDate = (int) Math.round(Math.floor(1 + 10 * Math.random()));
                        int penalty = (int) Math.round(Math.floor((quantity) * Math.floor(1 + 50 * Math.random())));


                        if(Math.random() < 0.5) {
                                type = "small";
                                screen = 5;
                                battery = 2000;
                        }
                        else {
                                type = "phablet";
                                screen = 7;
                                battery = 3000;
                        }

                        if(Math.random() < 0.5) {
                                ram = 4;
                        }
                        else {
                                ram = 8;
                        }

                        if(Math.random() < 0.5) {
                                storage = 64;
                        }
                        else {
                                storage = 256;
                        }

                        // Prepare the Query-IF message
                        ACLMessage msg = new ACLMessage(ACLMessage.PROPOSE);
                        msg.addReceiver(manufacturer);
                        msg.setLanguage(codec.getName());
                        msg.setOntology(ontology.getName());

                        // Prepare the content

                        phoneOrder.setQuantity(quantity);
                        phoneOrder.setUnitPrice(unitPrice);
                        phoneOrder.setDueDate(dueDate);
                        phoneOrder.setPenalty(penalty);
                        phoneOrder.setCustomer(myAgent.getAID());

                        Phone phone = new Phone();

                        phone.setType(type);

                        Screen SCRN = new Screen();
                        SCRN.setComponentName("Screen");
                        SCRN.setScreenSize(screen);
                        phone.setScreen(SCRN);

                        Storage STRG = new Storage();
                        STRG.setComponentName("Storage");
                        STRG.setStorageSize(storage);
                        phone.setStorage(STRG);

                        RAM RM = new RAM();
                        RM.setComponentName("RAM");
                        RM.setRamSize(ram);
                        phone.setRAM(RM);

                        Battery BTY = new Battery();
                        BTY.setComponentName("Battery");
                        BTY.setBatterySize(battery);
                        phone.setBattery(BTY);
```

```
phoneOrder.setPhone(phone);

                Action myOrder = new Action();
                myOrder.setAction(phoneOrder);
                myOrder.setActor(manufacturer);
                try {
                        // Let JADE convert from Java objects to string
                        getContentManager().fillContent(msg, myOrder);
                        send(msg);
                        //myAgent.addBehaviour(new CollectOffers(myAgent));


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

Figure 6

```
public class ReceiveDeliveryListener extends CyclicBehaviour
        {
                public ReceiveDeliveryListener(Agent a) {
                        super(a);
                }

                @Override
                public void action() {

                        MessageTemplate mt = MessageTemplate.and(MessageTemplate.MatchPerformative(ACLMessage.INFORM),
MessageTemplate.MatchConversationId("delivery"));
                        ACLMessage msg = myAgent.receive(mt);
                        if(msg != null)
                        {
                                try
                                {
                                        ContentElement ce = null;

                                        ce = getContentManager().extractContent(msg);
                                        if(ce instanceof Action)
                                        {
                                                Concept action = ((Action)ce).getAction();
                                                if(action instanceof Deliver)
                                                {
                                                        Deliver delivery = (Deliver)action;

                                                        int i = 0;

                                                        while(i < Orders.size()) {
                                                                if(Orders.get(i).getQuantity() == delivery.getQuantity() &&
(Orders.get(i).getQuantity() * Orders.get(i).getUnitPrice()) == delivery.getOrderCost()) {
                                                                        Orders.remove(i);
                                                                        i = Orders.size();
                                                                }
                                                                else {
                                                                        i++;
                                                                }
                                                        }
                                                }
                                        }
                                } catch (CodecException ce) {
                                        ce.printStackTrace();
                                }
                                catch (OntologyException oe) {
                                        oe.printStackTrace();
                                }
```

Figure 7

```
private class QueryBehaviour extends Behaviour{

	public QueryBehaviour(Agent a) {
		super(a);
	}

	int step = 1;

	int messagesReceived = 0;

	@Override
	public void action() {

		while(messagesReceived != 5) {
			step = 1;
			switch(step) {
			case 1:
				//This behaviour should only respond to QUERY_IF messages
				MessageTemplate mt =
					MessageTemplate.or(MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF),
					MessageTemplate.MatchPerformative(ACLMessage.INFORM));
				ACLMessage msg = receive(mt);
				if(msg != null){
					if(msg.getContent().contains("end of order")) {
						messagesReceived = 5;
					}
					else {
						try {
							ContentElement ce = null;
							// Let JADE convert from String to Java objects
							// Output will be a ContentElement
							ce = getContentManager().extractContent(msg);
							if (ce instanceof Owns) {
								Owns owns = (Owns) ce;
								Component it = owns.getComponent();
								Component compOrder = (Component)it;

								//check if seller has it in stock
								if(stockList.containsKey(compOrder.getComponentSize())) {

								ACLMessage response = new ACLMessage(ACLMessage.CFP);
									response.addReceiver(msg.getSender());
									response.setLanguage(codec.getName());
									response.setOntology(ontology.getName());
									response.setContent(msg.getContent());
									myAgent.send(response);
									step++;

								}


							}
							else {
								System.out.println("out of stock");

								ACLMessage response = new ACLMessage(ACLMessage.REFUSE);
								response.setContent(msg.getOntology());
								response.addReceiver(msg.getSender() );
								myAgent.send(response);

							}
						} catch (Exception e) {}
					}
				}
```

Figure 8

```
MessageTemplate sr = MessageTemplate.MatchPerformative(ACLMessage.PROPOSE);
ACLMessage msgS = receive(sr);
if(msgS != null){
        try {
                ContentElement ceS = null;

                // Let JADE convert from String to Java objects
                // Output will be a ContentElement
                ceS = getContentManager().extractContent(msgS);
                if(ceS instanceof Action) {
                        Concept action = ((Action)ceS).getAction();
                        if (action instanceof Sell) {
                                Sell order = (Sell)action;
                                Component itS = order.getComponent();
                                Component compOrderS = (Component) itS;

                                ACLMessage reply = new ACLMessage(ACLMessage.CONFIRM);
                                reply.addReceiver(msgS.getSender());
                                reply.setLanguage(codec.getName());
                                reply.setOntology(ontology.getName());

                                Component replyComponent = new Component();

                                replyComponent.setComponentName(compOrderS.getComponentName());
                                replyComponent.setComponentSize(compOrderS.getComponentSize());

                                Sell sellCompS = new Sell();
                                sellCompS.setBuyer(msgS.getSender());
                                sellCompS.setComponent(replyComponent);
                                sellCompS.setDeliveryDate(1);

                                sellCompS.setComponentPrice(stockList.get(compOrderS.getComponentSize()));
                                sellCompS.setQuantity(order.getQuantity());

                                Action request = new Action();
                                request.setAction(sellCompS);
                                request.setActor(msgS.getSender());

                                try {
                                        getContentManager().fillContent(reply, request);
                                        send(reply);
                                        messagesReceived++;

                                }
                                catch (CodecException ce2) {
                                        ce2.printStackTrace();
                                }
                                catch (OntologyException oe) {
                                        oe.printStackTrace();
                                }
                        }

                }
        } catch (Exception e) {}

}
```

Figure 9

```
openDeliveries.add(order);
```

Figure 10

```
if(openDeliveries.isEmpty() == false) {
        for(Sell temp : openDeliveries) {
                if (temp.getDeliveryDate() == day) {
                        System.out.println(temp.getComponent().getComponentName() + " x" + temp.getQuantity() + " delivered");
                        if(stock.containsKey(temp.getComponent().getComponentSize())){
                                stock.put(temp.getComponent().getComponentSize(),
stock.get(temp.getComponent().getComponentSize()) + temp.getQuantity());
                        }
                        else {
                                stock.put(temp.getComponent().getComponentSize(), temp.getQuantity());
                        }
                }
        }
        System.out.println("");
}
```

Figure 11

```
// Storage costs calculated here.

int sum = 0;
for (int f : stock.values()) {
        sum += f;
}

System.out.println("");
System.out.println("TODAY'S EXPENDITURE: ");
System.out.println("Order costs: -£" +  todaysCost);
System.out.println("Late fees: -£" + lateFees);
System.out.println("Warehouse cost: -£" + (5*sum));
System.out.println("Orders Delivered: +£" + (income + income02));

bank = bank + (income+income02) + (0-lateFees) + (0-5*sum) + (0-todaysCost);
```

Figure 12

```
OrderInst order = openOrders.get(indexes[n]);
                int quantity = order.getQuantity();

                int partOne = order.getPhone().getScreen().getScreenSize();
                int partTwo = order.getPhone().getStorage().getStorageSize();
                int partThree = order.getPhone().getRAM().getRamSize();
                int partFour = order.getPhone().getBattery().getBatterySize();


                // If the parts key is in the stock list
                if(stock.containsKey(partOne) && stock.containsKey(partTwo) && stock.containsKey(partThree) &&
stock.containsKey(partFour)) {

                        // If the parts quantity is in stock
                        if(stock.get(partOne) >= quantity) {
                                if (stock.get(partTwo) >= quantity) {
                                        if (stock.get(partThree) >= quantity) {
                                                if (stock.get(partFour) >= quantity) {
                                                        orderToBuild = indexes[n];
                                                        buildable = true;
                                                        break;

                                                }
                                                else {
                                                        break;
                                                }
```

Figure 13

```
public void action() {

        if(day != 1 && !openOrders.isEmpty()){

                int i = 0;

                if(ordersBuiltToday < 50){
                        while(i < openOrders.size()){
                                if((openOrders.get(i).getQuantity() + ordersBuiltToday) <= 50){
                                        OrderInst order = openOrders.get(i);

                                        int quantity = order.getQuantity();

                                        int partOne = order.getPhone().getScreen().getScreenSize();
                                        int partTwo = order.getPhone().getStorage().getStorageSize();
                                        int partThree = order.getPhone().getRAM().getRamSize();
                                        int partFour = order.getPhone().getBattery().getBatterySize();

                                        // If the parts key is in the stock list
```

Figure 14

```
// WORK OUT LATE FEES BEFORE ORDERS ARE BUILT AND REMOVED FROM OPEN ORDERS
        for (int i = 0; i < openOrders.size(); i++) {
                if(openOrders.get(i).getDueDate() < 0) {
                        lateFees = lateFees + openOrders.get(i).getPenalty();
                }
        }
```