Multi-Agent Systems Coursework Report

Abstract

This project uses agents to tackle a computing problem involving the supply chain of creating and selling mobile phones, where agents handle the roles of parts supplying, phone manufacturing and customers who order and buy the phones. The project implements a 100-day simulation of this agent supply chain, with a focus on the performance of the manufacturer agent, of which there is only one within the system. The system is then experimented with, to test hypotheses based on how the system will perform.

Introduction

This project tackles the "smartphone supply chain problem", a computing problem focused on the automation of the manufacture and sales of mobile phones. The problem takes place from the perspective of a mobile phone manufacturer, who needs to buy parts from their suppliers, take orders from customers and then construct telephones based upon their orders with the parts in stock. The crux of the problem is maximising profits and minimising squandered time through the use of computing solutions. This is important in the current market due to the ongoing "arms race" to automate every part of the supply chain, allowing businesses to be more efficient in their processing with the aim of landing higher net profits. The automation allows for a reduction in unnecessary staff as well as improving the efficiency and accuracy of tasks, such as finding parts at the best price from suppliers, at the cost of the lump sum development cost of a system to undergo the process.

There are various potential approaches to tackling the automation, however the one of interest with this project is using multi-agent systems. Groves, Collins, Gini and Ketter (2014) explore the idea of utilising agents for use within supply chain management. They suggest that agents can be very effectively utilised under the right market environments. They present the idea using their own set of agents with various simulated market conditions. They conclude that although their simulated market conditions did not match that of any real-world markets, valuable insights were gained to suggest strong potential of agents having usage within real-world market environments.

Model design

Agent role identification:

Manufacturer - The "main" agent within the system, of which there will only be one at any given time. This agent is tasked with receiving orders from customer agents, ordering parts from supplier agents, and optimising time and cost efficiency by planning how to tackle each order, and which suppliers to buy which parts from. This agent calculates all profits made and deducts all costs for parts and part storage to calculate net profit. A maximum of 50 ordered phones can be made within a single simulated day. When a customer's order is received, a total due date is provided for the order, if this date is not met, a penalty is also incurred.

Customer - Within the system, there will be three customer agents by default, which can be changed with a variable. For every day simulated within the system, each customer will generate one order per day. Each order will consist of only one type of phone (each having all the same parts) which will be randomly generated each day, in a random quantity, which is also randomised per day.

Supplier - There will be two suppliers within the system by default, each with their own inventory and parameters. The first supplier stocks every part required by the manufacturer and delivers ordered parts the next day. The second supplier only has around half of the parts that the manufacturer needs and takes four days to deliver, however the stocked parts are massively discounted in comparison to the first supplier.

Ontology justification:

The ontology used for the design of the system can be found within *Appendix 1*. The ontology can be broken down into three sections.

First is the "phone part" section, this is used to define what a part is, what parts there are, and what variations of each part are available. For example, within the system there are only two types of battery, 2000 and 3000 mAh variations, therefore the ontology shows specifically only those two types of battery.

The second section of the ontology is the "order" section, which breaks down what any single order from a single customer will contain. The things found here are the "phone type", which is the design specification of the type of phone being ordered, since only one type of phone is ordered at a time, the "number of devices", which is the amount of phones of the single type within the order, and the "due date", being the randomised date to which the order is due before late fees are applied.

The third and final section of the ontology is the sales chain, a section which shows which members of the supply chain sell to whom. It shows suppliers, who each have a name, a list of phone parts and a delivery time associated to them, selling to the manufacturer. It then shows the manufacturer, who in this system has exactly three customers and two suppliers associated with them, as well as having a list of orders, selling to a customer. The customer is the end of this sales chain, with each customer having a name and a list of orders they've made. In the case of the order lists held by both the manufacturer and by each customer, no minimum boundary is set, since both can have zero orders currently active at a given time.

Agent communication protocols:

The sequence diagram and ACL protocols created at this stage of the project can be seen within *Appendix 2*. It should be noted that the red text on the ACL protocols is space for a variable which depends on what the agents are requesting.

The customer element of the sequence diagram is broken into four interactions. The first is the customer generating an order and sending it to the manufacturer, the second and third are both receiving responses to their order (accepted or rejected), and lastly the manufacturer returning a message to inform the customer the order is completed. It should be noted that the customer creating the order is always the start of the process shown in the diagram, and them receiving their order is always the end. In the full system, this will run three times per day, one for each customer in the system.

The manufacturer element of the sequence diagram runs as a single interaction. This is because although parts will be ordered to replace those used, the order will still be attempted to be completed with parts currently stored in the warehouse. During the process of working on building the order, the manufacturer will always attempt to order parts to replace those used, since there is no benefit to ordering parts in bulk.

The supplier is also a single interaction, being to receive an order, confirm the order, prepare the order, then complete and send the order. The manufacturer can potentially order from two different suppliers at a time, depending on if the system determines ordering from both to be more beneficial than ordering from only one. The supplier has no rejection state in this system since the environment is unchanging, where suppliers each have an unlimited amount of stock for each of their items.

There is no fail state for the phones being built since the system also doesn't have the capacity to fail in such a way, only taking longer than intended, which incurs a "late fee".

Model implementation

Agents:

Within my implementation, there are four total agent types. There are the three agent types expected by the specification, being Manufacturer, Customer and Supplier, as well as a fourth agent type, "SystemTicker", which is tasked with synchronising each of the other agents in the system with the simulated 100-day cycle.

SystemTicker implements one behaviour, "SynchroniseAgents", which is used to keep each of the agents on track. Whenever a new day begins within the system, it sends out a "new day" message to each active agent it can find. This can be seen in *figure 1* of *appendix 3*. After sending "new day" messages to each of the agents, it awaits "done" responses from each of them. This can be seen in *figure 2* of *appendix 3*. Lastly, if the simulation has run for 100 days, instead of sending a "new day" message, the agent will send a "terminate" message to each of the other agents. This can be seen in *figure 3* of *appendix 3*.

All agents aside from SystemTicker each have one main Cyclic behaviour named "AwaitTicker", which relies on the SystemTicker telling them a day has started so they can each get along with their daily activities (or terminate if that is what is sent instead). This can be seen in *figure 4* of *appendix 3*. Each of the AwaitTicker behaviours also ends with a oneshot sub behaviour named “EndDay”, which handles the response of “done” to go back to SystemTicker to iterate the days once each agent has sent “done”. This can be seen in *figure 5* of *appendix 3*.

Customer implements six sub behaviours, each of the oneshot type. The first of these is “PrepareOrder”, a behaviour which calls a local method used to generate a single new order, to be used as that agent’s order for the day, applying each of the mathematical functions supplied in the coursework specification document. The next is “FindManufacturer”, a behaviour which locates the manufacturer agent for later use through the use of “DFService.search”. This can be seen in *figure 6* of *appendix 3*. Next is “SendOrder”, used to send the generated order to the located manufacturer. This can be seen in *figure 7* of *appendix 3*. Next is “AwaitResponse”, the behaviour tasked with receiving the acception or rejection of the order from the manufacturer. This can be seen in *figure 8* of *appendix 3*. Next is “DeliveryReceived”, used handle a delivery being received if one is sent. This can be seen in *figure 9* of *appendix 3*. Last is the EndDay behaviour required to function in the system.

Manufacturer implements six oneshot sub behaviours of its own. First is “NewDay”, a basic behaviour used to change days used to track when orders are due and receive any deliveries scheduled to be delivered on the current day. Next is “AwaitOrders”, another simple behaviour used to receive orders sent by each customer. This is followed by “DecideOrder”, which implements the strategy described in the “Design of manufacturer agent control strategy” section. “OrderParts” follows next, which locates “Supplier1” the supplier used for all order made with the control strategy implemented, then orders parts. The ordering can be seen in *figure 10* of *appendix 3*. “AssembleAndShipPhones” is next and is the other half of the control strategy. Lastly is the expected “EndDay” behaviour.

Last is Supplier, which implements two sub behaviours. First is “HandleRequests”, which receives all incoming parts requests, then responds to them with a generated invoice. This can be seen in *figure 11* of *appendix 3*. The other behaviour is the standard “EndDay” behaviour.

Ontology:

The ontology was expanded to five parts for the implementation, this is because the original version from the “Model Design” section was not detailed enough. There are two concepts, the First is “PhoneSpecification”, used to get 1 type of each part used together to make a phone. Second is “PartsList”, a list of each part type as an integer, implemented by manufacturer to store currentStock and requiredStock as singular variables. The other three parts are all predicates. “Order” uses PhoneSpecification alongside variables for each of the other elements of an order such as price per unit and quantity. It is used to send an order from a Customer to the Manufacturer. It can be seen in use in *figure 12* of *appendix 3*. Next is “PartsRequest”, implemented by Manufacturer to send an order to Suppliers. It implements a parts list as well as the Manufacturer’s Agent ID. It can be seen in use in *figure 13* of *appendix 3*. Lastly is “PartsInvoice”, sent back from the Supplier to the Manufacturer upon receiving a request for parts. The Invoice contains a list of parts that can be supplied by the supplier, alongside the price and date till delivery of the parts. It can be seen in use in *figure 14* of *appendix 3*.

Constraints:

1. Phone type constraint. This is met by using a method to generate each phone as specified in section 2.2 of the specification document. This can be seen in *figure 15* of *appendix 3*.
2. Component delivery times. Both of the suppliers have their own “info” classes, with all of the information pertaining to the specific supplier being stored to be loaded into a generic “supplier” agent. The info class of supplier1 can be seen in *figure 16* of *appendix 3*, the delivery being received by the manufacturer agent after the delivery wait is over can be seen in *figure 17* of *appendix 3*.
3. Component warehouse costs. Any items remaining in the “currentStock” inventory by the end of the day are automatically stored in the warehouse. Warehouse storage costs are calculated with [all of each part type multiplied programmed storage cost (w)]. This can be seen in *figure 18* of *appendix 3*.
4. Parts required for an order to ship. A total of 50 attempts can be made each day to build a phone, if the parts are not in stock, the phone will not be built but it will still count as an attempt. If all phones are built for an order, the order will be shipped. This can be seen in *figure 19* of *appendix 3*.
5. Maximum of 50 phones per day. This limit is enforced during the building process. It be observed alongside the parts requirement in *figure 19* of *appendix 3*.
6. Penalties for late delivery. Each day an order is in the possession of the manufacturer, a counter ticks down 1 day (starting at the specified days allocated to the order being completed by the customer). If the counter goes below 0 (-1 or lower), the equation [Specified Penalty multiplied by (days (less than 0 multiplied by -1)]. This can also be seen in *figure 19* of *appendix 3* alongside the implementations of constraints 4 and 5.
7. Profit calculation. Profit calculation is done throughout the process at each point where a monetary transaction would be made. There are two variables used, “todaysProfit” and “totalProfit”. Each day todaysProfit is reset to zero and is used to track the earnings of that day. totalProfit is appended to with the total of todaysProfit at the end of each day. todaysProfit is charged as part of the invoice seen in *figure 10* of *appendix 3*. Earnings from phone sales and losses from penalties are applied in the shipping behaviour of the agent, seen in *figure 19* of *appendix 3*. Warehouse costs are removed from todaysProfit before it gets appended to totalProfit at the start of the manufacturer’s “EndDay” behaviour. This can be seen in *figure 20* of *appendix 3*.

Design of manufacturer agent control strategy

Due to the time constraints of the project, only a simple control strategy was implemented. There was no researched theorem or academic literature behind the implemented strategy, however basic knowledge learned through education was applied.

The general idea behind the strategy was to minimise spending wherever possible and maximise profits from a single customer each day. This was done by only ever buying parts from Supplier1 due to the next day parts delivery, and taking only 1 order, desiring less than or exactly 50 phones, each day.

The accepted customer for each day would be the customer who would yield the highest profit of each customer who asked for 50 or less phones. This was done by looping through each customer and applying the equation of [(unit price of a single requested phone multiplied by total requested phones) minus (all parts in a single requested phone multiplied by total requested phones)]. There were checks in place to ensure no orders with more than 50 phones, nor orders which made no profit or less were accepted. If no orders were accepted for a day, no parts would be ordered for the next day, and no phone building would be done on the next day.

Components were all purchased from Supplier1 to be delivered and used the next day. This means no extra parts were ordered and no warehouse storage would be used under normal operating conditions. In the real world this would not work due to the myriad of potential errors, such as faulty parts, that could occur. However, this works in this system due to the lack of any potential mishaps of that sort.

Orders are all fully assembled and shipped the day after they are accepted, hence the reason for orders over 50 phones not being accepted.

This strategy allows for no late fees to ever be incurred, nor any warehouse costs. This means the only expenditure is that of the parts costs, allowing for maximum profits to be earned per order according to their specification.

This strategy is not flawless however and struggles in “bad luck” situations where the randomly generated orders only request lower amounts of phones or offer poor unit prices, as well as suffering in situations where two orders could have been taken for the next day for extra profits.

Experimental results

Experimental Plan:

Hypothesis; As the number of customer agents increase, so too does the chance of higher quality orders being received, meaning more profits will be earned and that the profits per set of 100 days would end up being more consistent than if ran with a smaller amount of customer agents. This will be tested using a range of 1-6 agents, each tested 20 times where the output Total Earnings after 100 simulated days will be taken as a metric. The agents tested with go no higher than 6 due to a noticeable drop in the performance of the simulation (each day takes multiple seconds to complete, when compared to the almost instantaneous day cycle seen with a lower amount of agents in the system) when 7 or more agents are used when running on the computer which all of the testing is done on.

Results:

Earnings in 100 simulated days per amount of customer agents simulated: (20 sets of 100 days per agent) [Commas added for table readability]					
1	**2**	**3**	**4**	**5**	**6**
£244,003	£411,134	£460,707	£562,060	£619,098	£584,669
£249,772	£374,031	£499,885	£607,477	£583,114	£706,089
£215,931	£388,663	£478,081	£517,149	£611,427	£633,141
£237,896	£426,083	£473,349	£523,800	£517,697	£695,572
£203,601	£371,086	£487,580	£639,764	£709,189	£687,237
£200,825	£362,894	£495,328	£528,709	£622,722	£758,560
£210,813	£404,686	£444,456	£602,471	£591,215	£621,183
£297,246	£340,961	£477,434	£518,218	£635,030	£688,899
£201,025	£330,612	£429,560	£558,552	£618,118	£695,425
£171,120	£328,343	£424,554	£540,470	£630,548	£634,860
£227,324	£384,102	£437,261	£494,232	£625,507	£585,799
£237,459	£316,391	£460,758	£547,036	£563,710	£625,295
£188,629	£340,305	£502,452	£530,624	£572,974	£658,933
£193,892	£341,852	£393,680	£621,522	£612,427	£716,880
£115,414	£374,178	£468,149	£568,491	£598,647	£738,401
£226,632	£412,877	£465,588	£598,605	£690,525	£697,500
£251,432	£374,240	£414,251	£529,996	£612,709	£616,721
£200,386	£401,190	£486,518	£514,102	£681,089	£668,991
£239,596	£417,395	£547,009	£488,072	£576,162	£595,819
£211,050	£357,603	£459,930	£571,929	£505,017	£629,813

Aggregated Results of 100 simulated days per amount of customer agents: (20 sets of 100 days per agent) [Commas added for table readability]						
	1	**2**	**3**	**4**	**5**	**6**
Mean (Average) earnings:	£216,202	£372,931	£465,327	£553,164	£608,846	£661,989
Highest earnings:	£297,246	£426,083	£547,009	£639,764	£709,189	£758,560
Lowest earnings:	£115,414	£316,391	£393,680	£488,072	£505,017	£584,669
Range [Highest-Lowest]:	£181,832	£109,692	£153,329	£151,692	£204,172	£173,891

Observations:

It can be observed that as the number of agents in the system increases, so to does average earnings, highest earnings and lowest earnings. It is worth noting that as more agents are added, the boost to earnings begins to plateau. If the scope of the testing were to span as far as 20-30 agents, this would likely become much more apparent.

Contrary to expectations when starting the text process, the range did not narrow as more agents were added. This is likely due to the relatively small scope of the tests done, getting only 20 sets of test data per agent. If this were changed to a far larger and more conclusive amount of sets such as 100 or 1000, I believe the narrowing of range would become visible.

Conclusions

As briefly touched upon earlier in this report, although the implementation chosen to tackle this problem was reasonably effective in the tackling of this problem as a coursework project, the implementation chosen would not have been suitable as a real-world solution to the problem. This is primarily because of a lack of “issues” arising within the confides of the coursework project, where the “issues” concerns real-world problems that could occur, such as a faulty shipment of parts, delivery delays, or supplier price changes. If touches such as these were added to the 100-day simulation, alongside methods of the manufacturer handling these issues, that would be a good first step in making the scenario more realistic. Another such touch could be having the customers attempt to amend their orders after sending them, further narrowing the gap between the simulation and a real-word scenario. Lastly, having the suppliers having a proper stock system instead of the “fantasy” of each supplier having infinite quantities of all stock in their inventory. This could lead to situations where the manufacturer may be unable to stock the required parts to complete an order.

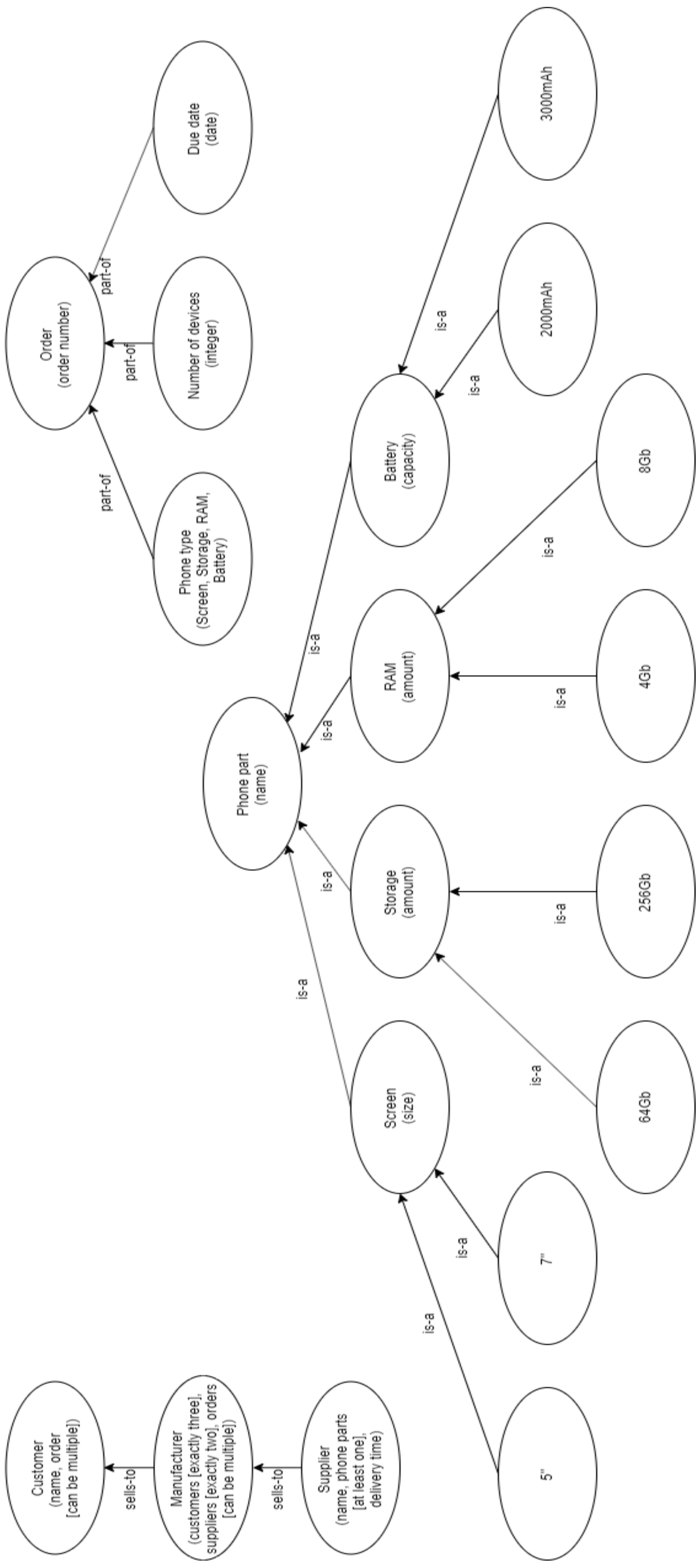
Within the scope of this project, the manufacturer strategy could also have been improved. The first, most basic improvement that could have been made would have been the ability to take more than one order per-day. This could still abide by the self-imposed rule of next-day order handling while also increasing the orders taken, subsequently increasing profits gained.

To further improve the manufacturer strategy, a full rework may be required. A much stronger idea for earning higher profits would be to stockpile orders with the highest possible profits per-unit, attempting to complete them as close to their deadlines as possible, as to still not incur late-delivery penalties.

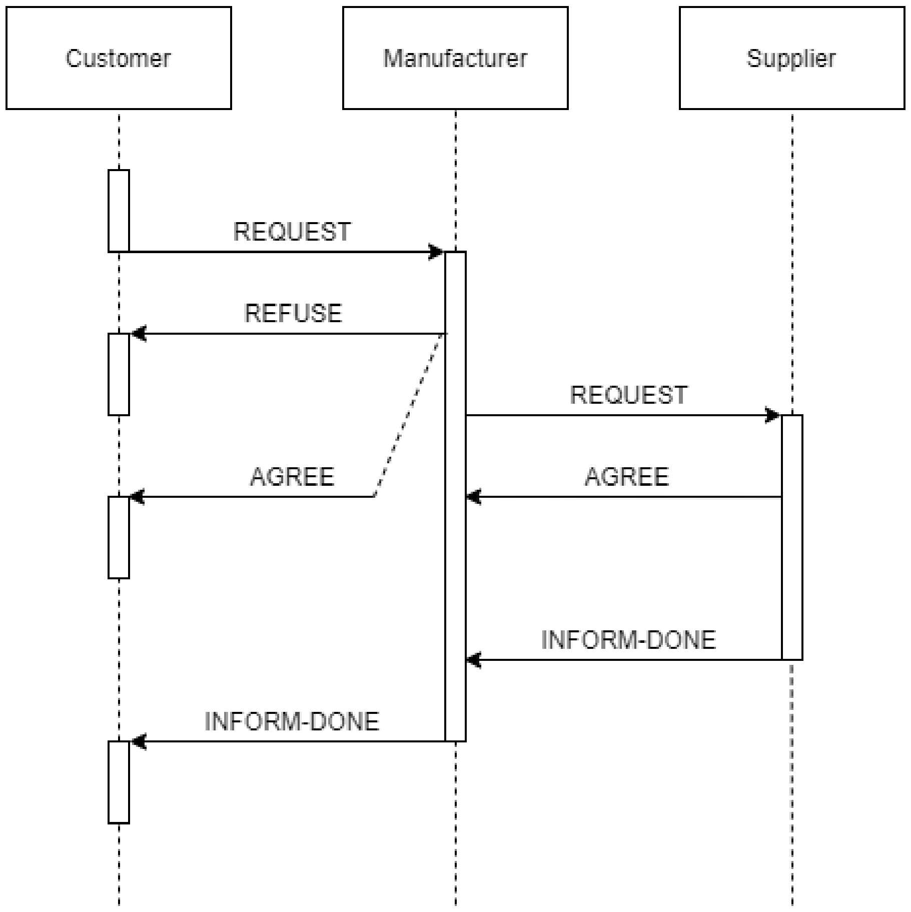
References

Groves, W., Collins, J., Gini, M., & Ketter, W. (2014). Agent-assisted supply chain management: Analysis and lessons learned. *Decision Support Systems, 57*(1), 274-284. doi:10.1016/j.dss.2013.09.006

Appendix 1: ontology


Order (order number)
Due date (date)
Number of devices (integer)
Phone type (Screen, Storage, RAM, Battery)
part-of
part-of
part-of
Phone part (name)
Battery (capacity)
RAM (amount)
Storage (amount)
Screen (size)
3000mAh
2000mAh
8Gb
4Gb
256Gb
64Gb
7"
5"
is-a
Customer (name, order [can be multiple])
Manufacturer (customers [exactly three], suppliers [exactly two], orders [can be multiple])
Supplier (name, phone parts [at least one], delivery time)
sells-to
sells-to

Appendix 2: communication protocols



```
(request
        :sender Customer
        :receiver Manufacturer
        :ontology SupplyChain
        :language fipa-sl
        :content
                action Manufacturer
                (orderPhone(:specification phoneSpecification))

(Refuse
        :sender Manufacturer
        :receiver Customer
        :ontology SupplyChain
        :language fipa-sl
        :content (order refuse))

(Agree
        :sender Manufacturer
        :receiver Customer
        :ontology SupplyChain
        :language fipa-sl
        :content (order confirm))

(inform-done
        :sender Manufacturer
        :receiver Customer
        :ontology SupplyChain
        :language fipa-sl
        :content (order completed & sent for delivery))

(request
        :sender Manufacturer
        :receiver Supplier
        :ontology SupplyChain
        :language fipa-sl
        :content
                action Supplier
                (orderParts(:parts requestedParts))

(Agree
        :sender Supplier
        :receiver Manufacturer
        :ontology SupplyChain
        :language fipa-sl
        :content (order confirm))

(inform-done
        :sender Supplier
        :receiver Manufacturer
        :ontology SupplyChain
        :language fipa-sl
        :content (order completed & sent for delivery))
```

Appendix 3: source code

```
ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
tick.setContent("new day");
for (AID agent : allAgents) {
        tick.addReceiver(agent);
}
myAgent.send(tick);
```

Figure 1: - “new day” message to all agents.

```
MessageTemplate mt = MessageTemplate.MatchContent("done");
ACLMessage msg = myAgent.receive(mt);
if (msg != null) {
        finishedMessages++;
        if (finishedMessages >= allAgents.size()) {
                step++;
        }
}
else {
        block();
}
```

Figure 2: - Awaiting “done” responses.

```
if (currentDay == simulationDays) {
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.setContent("terminate");

        for (AID agent : allAgents) {
                msg.addReceiver(agent);
        }

        myAgent.send(msg);
        myAgent.doDelete();
}
```

Figure 3: - “terminate” message being sent to all agents.

```
MessageTemplate mt = MessageTemplate.or(MessageTemplate.MatchContent("new ” +
        “day"), MessageTemplate.MatchContent("terminate"));
ACLMessage msg = myAgent.receive(mt);
if (msg != null) {
        if (systemTicker == null) {
                systemTicker = msg.getSender();
        }

        if (msg.getContent().equals("new day")) {
                //Do Daily SubBehaviours.
        }
        else {
                //Do anything that needs done before agent deletion.
                myAgent.doDelete();
        }
}
else {
        block();
}
```

Figure 4: - General AwaitTicker behaviour.

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(systemTicker);
msg.setContent("done");
myAgent.send(msg);
```

Figure 5: - “done” response.

```
DFAgentDescription manufacturerTemplate = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("manufacturer");
manufacturerTemplate.addServices(sd);

//Since there SHOULD only be one Manufacturer in the system, only the first
//found active manufacturer will be recorded.
try {
	DFAgentDescription[] manufacturers = DFService.search(myAgent,
		manufacturerTemplate);
	manufacturer = manufacturers[0].getName();
}
catch (FIPAException e) {
	e.printStackTrace();
}
```

Figure 6: - Code used to locate manufacturer for later use.

```
ACLMessage thisOrder = new ACLMessage(ACLMessage.REQUEST);
thisOrder.addReceiver(manufacturer);
thisOrder.setLanguage(codec.getName());
thisOrder.setOntology(ontology.getName());

try {
	getContentManager().fillContent(thisOrder, order);
	send(thisOrder);
}
catch (CodecException ce) {
	ce.printStackTrace();
}
catch (OntologyException oe) {
	oe.printStackTrace();
}
```

Figure 7: - Sending generated order to manufacturer.

```
while(!response){
	MessageTemplate mt;
	ACLMessage msg;

	mt = MessageTemplate.MatchPerformative(ACLMessage.AGREE);
	msg = receive(mt);
	if (msg != null) {
		if (msg.getContent().equals("order confirm")) {
			System.out.println(getAID().getLocalName()+
				": My Order was Accepted!");
			response = true;
		}
	}

	mt = MessageTemplate.MatchPerformative(ACLMessage.REFUSE);
	msg = receive(mt);
	if (msg != null) {
		if (msg.getContent().equals("order refuse")) {
			System.out.println(getAID().getLocalName()+
				": My Order was Rejected...");
			response = true;
		}
	}
}
```

Figure 8: - Handling for order being accepted or rejected.

```
MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
ACLMessage msg = receive(mt);
if (msg != null) {
	if (msg.getContent().equals("order completed")) {
		System.out.println(getAID().getLocalName()+
			": One of my orders has been delivered!");
	}
}
```

Figure 9: - Handling for an order being received.

```java
ACLMessage partsOrder = new ACLMessage(ACLMessage.REQUEST);
partsOrder.addReceiver(supplier1);
partsOrder.setLanguage(codec.getName());
partsOrder.setOntology(ontology.getName());

PartsRequest content = new PartsRequest();
content.setManufacturer(getAID());
content.setParts(requiredStock);

try {
	getContentManager().fillContent(partsOrder, content);
	send(partsOrder);

	MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.AGREE);
	ACLMessage msg = receive(mt);
	if (msg != null) {
		try {
			PartsInvoice invoice = (PartsInvoice)
				getContentManager().extractContent(msg);
			pendingDelivery.add(invoice);
			todaysProfit -= invoice.getCost();
			partsConfirmed = true;
			requiredStock = new PartsList();
		}
		catch (CodecException ce) {
			ce.printStackTrace();
		}
		catch (OntologyException oe) {
			oe.printStackTrace();
		}
	}
	else {
		block();
	}
```

Figure 10: - Sending a parts request and receiving a response.

```
MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
ACLMessage msg = receive(mt);
if (msg != null) {
	try {
		PartsRequest request = (PartsRequest)
			getContentManager().extractContent(msg);

		AID manufacturer = request.getManufacturer();
		PartsList requestedParts = request.getParts();

		PartsList invoiceParts = new PartsList();

		int totalCost = 0;

			//Generate invoice content

		PartsInvoice invoice = new PartsInvoice();
		invoice.setDueDays(deliveryDays);
		invoice.setCost(totalCost);
		invoice.setParts(invoiceParts);

		ACLMessage confirmOrder = new ACLMessage(ACLMessage.AGREE);
		confirmOrder.addReceiver(manufacturer);
		confirmOrder.setLanguage(codec.getName());
		confirmOrder.setOntology(ontology.getName());

		try {
			getContentManager().fillContent(confirmOrder, invoice);
			send(confirmOrder);
		}
		catch (CodecException ce) {
					ce.printStackTrace();
		}
		catch (OntologyException oe) {
			oe.printStackTrace();
		}
```

Figure 11: - Receive parts request then respond with an invoice.

```
private Order generateOrder() {
	//Create instance of Order class.
	Order thisOrder = new Order();

	//Name of Customer Ordering
	thisOrder.setCustomer(getAID());

	//Specification of phones being ordered
	thisOrder.setPhone(generatePhone());

	//Quantity of specified phone being requested.
	thisOrder.setQuantity((int) Math.floor(1 + (50*Math.random())));

	//Price of each phone specified.
	thisOrder.setPrice((int) Math.floor(100 + (500*Math.random())));

	//Number of days till order is due.
	thisOrder.setDays((int) Math.floor(1 + (10*Math.random())));

	//Penalty for late order delivery.
	thisOrder.setPenalty(thisOrder.getQuantity() * ((int) Math.floor(1 +
		(50*Math.random()))));

	//Return Order.
	return thisOrder;
}
```

Figure 12: - “Order” in use for generating an order to be sent to the manufacturer.

```
PartsRequest content = new PartsRequest();
content.setManufacturer(getAID());
content.setParts(requiredStock);
```

Figure 13: - Creating a “PartsRequest” to be sent to the supplier (requiredStock is an instance of PartsList).

```java
AID manufacturer = request.getManufacturer();
PartsList requestedParts = request.getParts();

PartsList invoiceParts = new PartsList();

int totalCost = 0;

//If the items requested are stocked by the supplier, add them to the invoice.
for (int i = 0; i < stock.length; i++) {
        if (stock[i].equals("5\" Screen")){
                invoiceParts.setScreen_5inch(requestedParts.getScreen_5inch());
                totalCost += prices[i] * requestedParts.getScreen_5inch();
        }
        else if (stock[i].equals("7\" Screen")) {
                invoiceParts.setScreen_7inch(requestedParts.getScreen_7inch());
                totalCost += prices[i] * requestedParts.getScreen_7inch();
        }
        else if (stock[i].equals("64Gb Storage")) {
                invoiceParts.setStorage_64Gb(requestedParts.getStorage_64Gb());
                totalCost += prices[i] * requestedParts.getStorage_64Gb();
        }
        else if (stock[i].equals("256Gb Storage")) {
                invoiceParts.setStorage_256Gb(requestedParts.getStorage_256Gb());
                totalCost += prices[i] * requestedParts.getStorage_256Gb();
        }
        else if (stock[i].equals("4Gb RAM")) {
                invoiceParts.setRAM_4Gb(requestedParts.getRAM_4Gb());
                totalCost += prices[i] * requestedParts.getRAM_4Gb();
        }
        else if (stock[i].equals("8Gb RAM")) {
                invoiceParts.setRAM_8Gb(requestedParts.getRAM_8Gb());
                totalCost += prices[i] * requestedParts.getRAM_8Gb();
        }
        else if (stock[i].equals("2000mAh Battery")) {
                invoiceParts.setBattery_2000mAh
                        (requestedParts.getBattery_2000mAh());
                totalCost += prices[i] * requestedParts.getBattery_2000mAh();
        }
        else if (stock[i].equals("3000mAh Battery")) {
                invoiceParts.setBattery_3000mAh
                        (requestedParts.getBattery_3000mAh());
                totalCost += prices[i] * requestedParts.getBattery_3000mAh();
        }
}

PartsInvoice invoice = new PartsInvoice();
invoice.setDueDays(deliveryDays);
invoice.setCost(totalCost);
invoice.setParts(invoiceParts);
```

Figure 14: - Creating a “PartsInvoice” to be sent to the manufacturer as a response.

```java
private PhoneSpecification generatePhone() {
	PhoneSpecification phone = new PhoneSpecification();

	// Generate a random number.
	double randomNumber = Math.random();

	if (randomNumber < 0.5)
	{
		phone.setScreen(0);
			phone.setBattery(0);
	}
	else
	{
		phone.setScreen(1);
		phone.setBattery(1);
	}

	//	Generate a new random number.
	randomNumber = Math.random();

	if (randomNumber < 0.5)
	{
		phone.setRAM(0);
	}
	else
	{
		phone.setRAM(1);
	}

	//	Generate another new random number.
	randomNumber = Math.random();

	if (randomNumber < 0.5)
	{
		phone.setStorage(0);
	}
	else
	{
		phone.setStorage(1);
	}

	//Return randomly generated specification.
	return phone;
}
```

Figure 15: - New Phone specification generation method.

```
public class Supplier1_info {

	private String[] items = {"5\" Screen", "7\" Screen", "2000mAh Battery", 
		"3000mAh Battery", "4Gb RAM", "8Gb RAM", "64Gb Storage", 
		 "256Gb Storage"};
	private int[] prices = {100,150,70,100,30,60,25,50};
	private int deliveryDays = 1;

	public String[] Items() {
		return items;
	}

	public int[] Prices() {
		return prices;
	}

	public int DeliveryDays() {
		return deliveryDays;
	}
}
```

Figure 16: - Supplier1_info, including delivery time (deliveryDays).

```
if (pendingDelivery.size() > 0) {
	for (int i = 0; i < pendingDelivery.size(); i++) {
		pendingDelivery.get(i).setDueDays
			(pendingDelivery.get(i).getDueDays() - 1);
	}

	ArrayList<PartsInvoice> deliveryToday = new ArrayList<>();
	for (PartsInvoice invoice : pendingDelivery) {
		if (invoice.getDueDays() == 0) {
			deliveryToday.add(invoice);
			currentStock = combinePartsLists
				(currentStock, invoice.getParts());
		}
	}
	pendingDelivery.removeAll(deliveryToday);
}
```

Figure 17: - Deliveries being received by the manufacturer, where “pendingDelivery” is a list of all invoices received which have not yet had their contents delivered.

```
private int todaysWarehouseCost () {
      //Cost is all parts added together, multiplied by "storageCost" (w).
      int cost = 0;

      cost += currentStock.getScreen_5inch();
      cost += currentStock.getScreen_7inch();
      cost += currentStock.getBattery_2000mAh();
      cost += currentStock.getBattery_3000mAh();
      cost += currentStock.getRAM_4Gb();
      cost += currentStock.getRAM_8Gb();
      cost += currentStock.getStorage_64Gb();
      cost += currentStock.getStorage_256Gb();

      cost = cost*storageCost;

      return cost;
}
```

Figure 18: - Overnight parts storage cost.

```
int canBuildToday = 50;
int ordersRemainingPhones = todaysOrder.getQuantity();

while (todaysOrder.getCustomer() != null && canBuildToday != 0){

    boolean allPartsAvailable = true;
    PhoneSpecification currentPhone = todaysOrder.getPhone();

    // Same case as in "DecideOrders", can be easily modified if there are
    //     more than two types of any given part.
    if (currentPhone.getScreen() == 0) {
        if (currentStock.getScreen_5inch() != 0) {
            currentStock.setScreen_5inch
                (currentStock.getScreen_5inch() -1);
        }
        else {
            allPartsAvailable = false;
        }
    }

    //Repeated for all part types with if… else (cut down in REPORT for the
    //sake of saving space).


    if (allPartsAvailable) {
        ordersRemainingPhones--;
    }

    //Iterates down since an attempt was made to get a phone built.
    canBuildToday--;

    if(ordersRemainingPhones == 0) {
        //Send message to Customer that their order has been sent.
        //(Removed in REPORT for the sake of saving space).

        todaysProfit += (todaysOrder.getPrice() *
            todaysOrder.getQuantity());

        //Apply late fees to phones with less than 0 days remaining.
        if (todaysOrder.getDays() < 0) {
            todaysProfit -= (todaysOrder.getPenalty() *
                (todaysOrder.getDays()* -1));
        }

        //Remove order 0 (todaysOrder) as it has been handled.
        allOrders.remove(0);

        //Get rid of todaysOrder as it has been completed, allowing the
        // loop to end early if it has not yet hit 50 phones built today.
                    todaysOrder = new Order();
                }
            }
        }
```

Figure 19: - Phones being built and shipped once all phones are done, including the 50 limit and penalty application.

```
//Apply overnight parts storage costs.
todaysProfit -= todaysWarehouseCost();
totalProfit += todaysProfit;

//Display the earnings for this day.
System.out.println("Todays earnings: £" + todaysProfit);

//Display total earnings over the whole simulation.
System.out.println("Total earnings: £" + totalProfit);

//Set all daily variables back to default.
todaysCustomers = 0;
todaysProfit = 0;
step = 0;
partsConfirmed = false;

ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(systemTicker);
msg.setContent("done");
myAgent.send(msg);
```

Figure 20: - Manufacturer's "EndDay" with profit calculations.