Introduction

In today’s competitive market, it can be extremely difficult for businesses to survive let alone thrive at any stage of the supply chain. The supply chain is made up of 3 core components. Customers with the intent to purchase finished goods, manufacturers who assemble the finished goods from a set of materials and the suppliers who source materials to sell to manufacturers. There may be other lower level suppliers who sell pure or raw materials to higher level suppliers. In order to compete with competitors at all stages of the supply chain, supply chain management is extremely important in order to meet market demands from customers while keeping up with changes in trends/practices[ins-ref]. Managing a supply chain efficiently begins with effective planning, sourcing, coordination and procurement. Performing all stages well is crucial to the success of the supply chain model [ins ref]. Supply chain businesses are at the forefront of the worldwide economy accounting for trillions in yearly transactions. This cements the supply chain problem as a major pillar not only in the world’s economy but also in the lives of the millions of people employed by the supply chain. Agent based programming provides the perfect environment for testing/evaluating different supply chain strategies. Each part of the chain can be represented as one or more agents designed to replicate the supply chain as it is in the real world. Intelligent agents, unlike classes in object orientated programming are designed to “think” or make decisions based on the information provided to them either by other agents or information sourced from the simulation world. For example, a manufacturer agent would be able to decide whether to accept an order based on its profitability or a supplier agent would set its pricing to be competitive with other supplier agents to entice the manufacturer agents to purchase from it. As agents make decisions for themselves autonomously the study of how agents react under changes in circumstance can provide a valuable insight into how to deal with such situations in the real world without the associated risks. Identifying strategies to improve performance, whether that be customer satisfaction or profit generated or another metric, can give a glimpse into how to stay competitive in the market [INSREF].

Model Design

To properly demonstrate the problem domain within my application I first had to consider the types of agent I would need to properly represent the supply chain like it is in the real world. When deciding on my agent choices I looked to represent each part of the problem, without overly complicating it with an abundance of unnecessary agents that carry out simple jobs which could be incorporated into a sub job inside one of my other agents. Below is a subsection for each agent which contains information about them and their duties or responsibilities within the simulation environment.

1. Customer Agent

The customer agent is used to replicate any number of different customers and can be instantiated as many times as required. This means my application can be used to simulate a supply chain of multiple sizes depending on the number of customers used, as the more customers in the simulation the more orders there are sent to the manufacturer. Each customer is designed to submit 1 order per day to the manufacturer agent as specified in the requirements. The quantity of phones along with the specification of each phone is randomly generated through a series of calculations which generate a random number. The phone ordered by the customer depends on the number generated for each part. Once the phone specification is created it is sent along with the quantity to be made, when the orders due, per day late fee and an order ID to the manufacturer agent in the form of a message. In JADE, messages adhere to the Agent Communication Language (ACL) which allows for encoding of content to be sent between agents. Specifically, JADE uses the FIPA SL which is used for the encoding of concepts, actions and predicates. As shown in my communication protocol diagram (Appendix 2) I chose to use a REQUEST message format for this communication within my ontology which deals with all communications involving the sending of an order. It also makes the most sense in terms of the simulation as the manufacturer can choose to accept or decline the order in response.

1. Manufacturer Agent

The manufacturer agent is used to replicate the duties of the manufacturer in the supply chain. It is used to receive and process the orders sent from customer agents. The manufacturer also acts a middleman of sorts between the supplier agents and the customer agents. When an order is received by my manufacturer agent It will first check my warehouse to see if it has all the parts required to make the order. If there is insufficient stock send a request to the supplier for the parts to make the order. The manufacturer agent is also limited to creating 50 phones per day which can lead some orders not being made on certain days. Once the supplier provides the parts for an order the parts are added to the warehouse. The warehouse in my manufacturer agent is represented as a hash map which contains the part name as a key and the quantity of said part as a value. Once the parts are present in the warehouse the order is assembled and delivered to the customer agent through a REQUEST message (Appendix 2) containing the order of smartphones.

1. Supplier Agent

The purpose of the supplier agents in the simulation is to provide the parts to the manufacturer required to create the orders for the customer agents. In our simulation there is two supplier agents which act in the same way. When an order is sent as a request message (Appendix 2) from the supplier the supplier will provide these parts needed for the order in exchange for money taken from the manufacturer’s currentMoney which is stored as an integer variable. The pricing for each part was provided to us in the problem requirements. Supplier 2 offers cheaper pricing for RAM and Storage at the cost of not selling the other components along with a much longer delivery time of 4 days. This is 4 times longer than supplier 1 which delivers the next day. Unlike the manufacturer the supplier will accept every order of goods presented to it as the task for this coursework was to maximise the profit of the manufacturers, not the suppliers.

1. Ticker Agent

The final type of agent within my program is the Ticker agent. This can be thought of as the “Controller” of the system which manages when days start and end within the simulation. The ticker agent is also responsible for adding all the agents to the directory services. Each day every other agent present in the situation sends an INFORM message to the ticker agent (Appendix 2) simply containing “done”. This message is sent from each agent once their daily activities are completed. Once the ticker agent has received a done INFORM message from all of the agents present in the simulation it will trigger a new day and send a new INFORM message to all agents informing them that a new day has begun. Once the other agents receive this method they will start their daily activities. The ticker agent also controls the total amount of days the simulation is ran for.

1. Ontology

When creating my ontology my aim was to produce an easy to follow, logical approach which encapsulated the supply chain problem. My first thought was to create a phone concept which contained everything any agent would have to know about a particular phone. As you can see from my diagram (Appendix 1) this includes the serial number of that phone, the quantity of that phone needed for the order, the due date of that phone and the late delivery cost per day. As each order contains multiple phones of the same type I created an order concept to contain the specification of each phone, quantity of each phone and the price to the customer for the order. (Appendix 1) .To store the specification for the phones of that order I decided to implement a concept known as item. The item class itself contains no code but it is implemented as part of my order class in the form of an array list to store a list of items (Appendix 1). This list of items contains the specification for phones from that order. This allows me to easily find out what’s needed for each order. Each type of component, RAM, storage, battery and screen all extend from the Item concept (Appendix 1). One of the most common actions that the agents within the simulation have to perform is to send an order. Whether that be a customer sending an order to the manufacturer, manufacturer sending an order to a supplier etc. In JADE data types must be wrapped by an action. For sending orders I created the SendOrder Action (Appendix 1) which contains the AID of the destination agent and the order being sent. I believe my ontology is effective as it is simple but allows the agents to communicate well with one another. Due to its simplicity the ontology is very scalable and could be used by someone with no inside knowledge of how the agents in the simulation run.

Model Implementation

Customer Agent

|  |  |  |  |
| --- | --- | --- | --- |
| Behaviour Name | Type of behaviour | Overview | Communication |
| findManufacturers | OneShotBehaviour | Searches through the df to find all manufacturer agents and adds their AID to a list named manufacturers. | No communication |
| sendOrders | OneShotBehavior | Calls the generateOrder method to create the order. Sends order to manufacturer Agent. | Request Message which sends the order to the manufacturer using the conversation ID “cust-order” |
| recieveOrders | SimpleBehaviour | Constantly listens for incoming fulfilled orders from the manufacturers. | Listening for incoming request messages with the conversation ID “completed-order”. |
| endDay | OneShotBehaviour | Informs the ticker agent that the customer has completed all its daily activity. | Inform message |

Manufacturer Agent

|  |  |  |  |
| --- | --- | --- | --- |
| Behaviour Name | Type of behaviour | Overview | Communication |
| findCustomers | OneShotBehaviour | Search through the DF to find agents of type customer and add them to an AID list known as customers. | No communication |
| recieveOrders | SimpleBehaviour | Receives incoming orders from customers. | Listens for incoming Request messages with the Conversation ID “cust-order”. |
| purchaseParts | OneShotBehaviour | Sends the order containing the parts to be ordered to one of the supplier agents. | Sends a request message to a supplier with the Conversation ID “manufacturer-order”. |
| recieveParts | SimpleBehavior | Receives parts for an order from the supplier agent. It then sorts each part into the hashmap warehouse. The cost of each item is also deducted from the manufacturers current money. | Listens for incoming Request messages with the Conversation ID “supplier-parts”. |
| sendOrders | OneShotBehaviour | Compares quantity of parts in warehouse to parts required to make order. If the warehouse contains all parts for the order the order is then shipped to the Agent. | Sends a message containing the finished order to the customer agent using the Conversation ID  “completed-order” |
| WareHouseTax | OneShotBehaviour | Calculates todays warehouse tax to be deducted from profit by charging £5 per item in the warehouse. | No communication. |
| endDay | OneShotBehaviour | Sends a message to the ticker agent informing it that the manufacturers daily activities are complete. | Inform message |

Supplier Agent

|  |  |  |  |
| --- | --- | --- | --- |
| Behaviour Name | Type of behaviour | Overview | Communication |
| recieveOrders | SimpleBehaviour | Receives orders from the manufacturer containing parts to be sent back. | Listens for incoming request messages with the Conversation ID “manufacturer-order”. |
| sendParts | OneShotBehaviour | Sends the parts required for that order to the manufacturer. | Sends a request message containing the parts for the order using the Conversation ID “supplier-parts”. |
| endDay | OneShotBehaviour | Sends a message to the ticker agent informing it that the suppliers daily activities are complete. | Inform Message. |

My agent action sendOrder is used in every request message sent as every request message sent within my simulation always has an order to be sent along with it. See examples in Appendix 3.

Constraints

Out of the 7 constraints specified in the coursework specification I managed to correctly implement 6. The only constraint not present within my program is the late delivery fee penalty charged to the manufacturer when an order is not sent to the customer by a certain day.

1. I make sure that the correct parts are chosen for either phone by specifying in my generateOrders() function which creates each build of phone that parts for phablets cannot be used for small smartphones and vice versa (Appendix 3 -4)
2. I ensure each supplier sends parts after the correct amount of days by using a simple counter to count the number of days which have passed. Once the delivery day is met the order is sent and the counter is simply reset to 0 (Appendix 3 – 5)
3. For my warehouse I use a hashmap which contains a string key which is the name of the item and an integer entry which represents the number in stock. I loop through my hashmap and count the number of entry’s. I then multiply this value by 5 to ger the correct costs for today(Appendix 3 – 5).
4. To check if the warehouse contains the correct components to create an order I loop through my current order stored in a list called ItemList and check if the instance of that list is either of type screen, ram, storage or battery. I then check the value and up the associated counter value for that item if its in the order. I then check if the quantity in the warehouse is less or equal, if so I plus 1 to my counter partsInStock. As there is 4 parts of the smartphone that must be in stock to create one complete phone I check if partsInStock is equal to 4 before the order is created(Appendix 3 – 6).
5. To make sure no order is made if it the quantity goes over 50 I simply use an integer variable named phonesMade to keep track of the amount of phones made that day. Before the order is sent in a message I check that the order quantity plus the number of phones already made. Then once the order is being sent I add the quantity of that order to the phonesMade integer (Appendix 3 – 6).
6. I correctly calculate the profit for the day in my endDay sub behaviour in my manufacturer by taking the manufacturers current balance and subtracting the warehouse costs. Note since my application does not include the late penalties constraint, I do not take this into consideration in the calculation of profit.

Design of manufacturer agent control strategy

1. To decide what orders to accept or not I decided to calculate the minimum amount of money an order would have to provide for it to be profitable. To do this I simply calculated the cost to make a standard smartphone and added some extra on top as profit. The value I decided to use was 350. Most orders are over this so the application does not deny too many orders (Appendix 3 – 11).
2. Even though the RAM and Storage components are slightly cheaper from supplier 2 I believe the increased delivery time is not worth the price difference. Had my simulation included the late delivery time fees this would have made this strategy even more effective but even without it I still believe it is the right decision. When considering the simulation like the real world, the slight difference in price could prevent customers from receiving their order in a timely manner which could cause them not to return to the manufacturer.
3. My application will purchase the parts for the order when it receives it and places the items into the warehouse. Depending on what order is made first there may still be items left in the warehouse so the warehouse is still checked for parts before an order is made so it is not assumed that the correct parts are there. I use this method as it should ensure that I don’t end up with massive warehouse costs that could arise through buying a lot of components at once to store in the warehouse. If a small quantity of phones is ordered after this it could leave the warehouse too well stocked for a large period of time which would negatively affect profit.
4. To decide what orders to create/ship my application will simply create the order which corresponds to the first shipment of goods from the supplier. This prevents any order waiting too long to be assembled and ensures that the customer will receive the order in the most time efficient way possible. This also should ensure that the warehouse does not get full of parts for orders that are going to be waiting several days until they are fulfilled.

Experimental Results

To test the performance of my simulation I will alter the number of customers for each of my set of runs. I will run my simulation 30 times with one customer, 30 times with two customers and 30 times with 3 customers to see if and how it affects the profit generated at the end of 100 days. I chose to alter this as it will give me the best indication of how my program performs as the environment changes with more customers placing orders each day.

Hypothesis – As the number of active customers in the simulation increases the profit generated should also increase proportionally as more orders will be sent to the manufacturer.

However, increasing the amount of customers will increase the amount of goods being stored in the warehouse which will negatively effect profit. Also if my simulation correctly implemented late fees the amount of late fees to be paid would also increase.

As you can see from the graph all the test cases produce fairly similar results regardless of number of customers. Each test was conducted 30 times for a duration of 100 simulated days. The mean profit of each manufacturer agent is as follows, 1 customer returned 884782, 2 customers returned 885724, 3 customers returned 895777. This shows my hypothesis was correct as the average profit does increase as the number of customers increases. However, I did expect more variation in the profit levels between the number of customers. This could indicate that my warehouse strategy is not optimal as the lack of difference could be put down to too much being stored in the warehouse when the amount of customers increases.

Conclusions

To conclude, my implementation of the agent strategy represents the supply chain strategy well with the appropriate types of agents, use of concepts and agent actions and a methodical strategy for generating profit. However to make the environment more realistic so that it could potentially be used by supply chain companies looking to test out new strategies to increase profit. One of these would be to implement a warehouse agent that’s daily activities would include listening for part requests from the manufacturer, sending a certain amount of parts to the agent depending on how many orders the manufacturer has received that day and calculating the warehouse costs. The warehouse agent could also implement its own strategies for buying stock from suppliers based on how much stock it usually holds and how long it holds it for. To expand on this idea, another way in which I could improve my simulation would be to implement a training mechanism of some kind to train each agent by rewarding them for having good practices. E.g. the manufacturer for making profit, the warehouse agent for keeping stock low while fulfilling all orders etc. One way this could be accomplished is through the implementation of an evolutionary algorithm designed to run alongside the system while producing generations of agents that gradually get more and more effective. It would also improve the autonomy of the agents as changes in the environment would be detected as the fitness level of each agent drops requiring the evolutionary algorithm to find another approach to improve its fitness scores. Another way I would create a more realistic simulation would be to implement a postal agent that would handle all the delivery times for each of the suppliers. This could be used to expand the number of suppliers to a much larger amount than the two used currently as the postal agent would be able to keep track of the varying delivery times. More manufacturer agents could also be added to increase profits generated as more orders could be produced each day. The postal agent would also keep track of which manufacturer has ordered what and when its due. It could also adopt a strategy where it would keep track of the due date on an order, as the due date draws closer it will inform the manufacturer that it should ship this order as soon as the part arrives to avoid fees. Adding an incentive for customers to buy more phones could also help boost profits. For example a slightly cheaper price could be offered to customers who order 25 or more phones. The simulations ontology could be expanded to account for this by creating a wholesale concept. This however could also improve the volume of phones to be made which would increase warehouse costs. In a real-world environment different parts of the supply chain will negotiate with one another to help improve their respective goals. This could be added into the simulation by way of exclusivity deals or goodwill. A supplier for example, may offer a manufacturer cheaper prices than its competitors but it is resigned to using that manufacturer for a set amount of time.

* Implement warehouse agent
* Postal agent to handle deliveries
* Discount on bulk buying
* Negotiation between agents
* Wider variety of products

Appendix 1

Appendix 2

Appendix 3

1.endDay sub behaviour from the customer agent

private class endDay extends OneShotBehaviour {  
 public endDay(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 ACLMessage msg = new ACLMessage(ACLMessage.*INFORM*);  
 msg.addReceiver(tickerAgent);  
 msg.setContent("done");  
 myAgent.send(msg);  
 }  
  
  
}

2.recieveOrders sub behaviour from the manufacturer agent

public class recieveOrders extends SimpleBehaviour {  
 public recieveOrders(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 MessageTemplate mt = MessageTemplate.*and*(MessageTemplate.*MatchConversationId*("cust-order"), MessageTemplate.*MatchPerformative*(ACLMessage.*REQUEST*));  
 ACLMessage msg = myAgent.receive(mt);  
 if (msg != null) {  
 no\_of\_customers++;  
 try {  
 ContentElement ce = null;  
 ce = getContentManager().extractContent(msg);//ERROR  
 Action available = (Action) ce;  
 SendOrder = ((SendOrder) available.getAction());// this is the order requested  
 System.*out*.println("Manufacturer has received: Order ID: " + sendorder.getOrder());  
 phonesToCreate.add(phone);  
  
 order = sendorder.getOrder();  
 phone = order.getPhone();  
 order.setPhoneOrderQuantity(phone.getQuantity());  
  
 currentMoney = currentMoney += order.getOrderCost();  
  
 itemList = order.getParts();  
  
 for (Item parts : itemList) {  
  
 if (parts instanceof Storage) {  
 storageList.add(((Storage) parts).getSpace());  
 }  
 if (parts instanceof Ram) {  
 ramList.add(((Ram) parts).getSize());  
 }  
  
  
 customer = sendorder.getCustomer();  
 }  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe) {  
 oe.printStackTrace();  
 }  
  
  
 }  
 }  
  
 public boolean done(){  
 if (no\_of\_customers == 3){  
 no\_of\_customers = 0;  
 return true;  
 }else{  
 return false;  
 }  
 }  
}

3.sendParts sub behaviour from the supplier agent

private class sendParts extends OneShotBehaviour {  
 public sendParts(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 if(daycounter == 1){  
 daycounter = 0;  
 }  
 if (itemList != null) {  
 DFAgentDescription manufacturerTemplate = new DFAgentDescription();  
 ServiceDescription sd = new ServiceDescription();  
 sd.setType("manufacturer");  
 manufacturerTemplate.addServices(sd);  
 try {  
 DFAgentDescription[] agentsType1 = DFService.*search*(myAgent, manufacturerTemplate);  
 for (int i = 0; i < agentsType1.length; i++) {  
 manufacturer.add(agentsType1[i].getName()); // this is the AID  
  
 }  
  
 } catch (FIPAException e) {  
 e.printStackTrace();  
 }  
  
  
  
  
 ACLMessage reqOrd = new ACLMessage(ACLMessage.*REQUEST*);  
  
 reqOrd.addReceiver(manufacturer.get(0));  
 reqOrd.setLanguage(codec.getName());  
 reqOrd.setOntology(ontology.getName());  
  
  
 reqOrd.setConversationId("supplier-parts");  
 SendOrder = new SendOrder();  
 sendOrder.setCustomer(this.myAgent.getAID());  
 sendOrder.setOrder(order);  
  
  
  
  
  
  
 Action request = new Action();  
 request.setAction(sendOrder);  
 request.setActor(manufacturer.get(0));  
  
 System.*out*.println("Supplier has sent parts requested for: Order ID: " + sendOrder.getOrder());  
 try {  
 getContentManager().fillContent(reqOrd, request); //send the wrapper object  
 send(reqOrd);  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe){  
 }  
  
 daycounter++;  
 }  
  
  
 }  
 }  
}

4.Correct phone parts for each type of phone.

public void generateOrders() {  
  
 if (Math.*random*() < 0.5) {  
 //small phone  
 battery.setCapacity(2000);  
 itemList.add(battery);  
 screen.setLength(5);  
 itemList.add(screen);  
 } else {  
 //phablet  
 battery.setCapacity(3000);  
 itemList.add(battery);  
 screen.setLength(7);  
 itemList.add(screen);  
 }  
 if (Math.*random*() < 0.5) {  
 ram.setSize(4);  
 itemList.add(ram);  
 } else {  
 ram.setSize(8);  
 itemList.add(ram);  
 }  
 if (Math.*random*() < 0.5) {  
 storage.setSpace(64);  
 itemList.add(storage);  
 } else {  
 storage.setSpace(256);  
 itemList.add(storage);  
 }  
  
  
 phone.setSerialNumber();  
 phone.setQuantity(Math.*floor*(1 + 50 \* Math.*random*()));  
 phone.setPricePerUnit(Math.*floor*(100 + 500 \* Math.*random*()));  
 phone.setnumDaysDue(Math.*floor*(1 + 10 \* Math.*random*()));  
 phone.setPerDayPenalty(phone.getQuantity() \* Math.*floor*(1 + 50 \* Math.*random*()));  
 phonesToBuy.add(phone);  
 System.*out*.println("Customer ordered: " + phone.getQuantity() + ", order due in " + phone.getNumDaysDue() + " days.");  
 ordersSent++;  
}

5.Component Delivery times are correct (supplier 1)

private class sendParts extends OneShotBehaviour {  
 public sendParts(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 if(daysPassed == 1) {  
 daysPassed = 0;  
 if (itemList != null) {  
 DFAgentDescription manufacturerTemplate = new DFAgentDescription();  
 ServiceDescription sd = new ServiceDescription();  
 sd.setType("manufacturer");  
 manufacturerTemplate.addServices(sd);  
 try {  
 DFAgentDescription[] agentsType1 = DFService.*search*(myAgent, manufacturerTemplate);  
 for (int i = 0; i < agentsType1.length; i++) {  
 manufacturer.add(agentsType1[i].getName()); // this is the AID  
  
 }  
  
 } catch (FIPAException e) {  
 e.printStackTrace();  
 }  
  
  
 ACLMessage reqOrd = new ACLMessage(ACLMessage.*REQUEST*);  
  
 reqOrd.addReceiver(manufacturer.get(0));  
 reqOrd.setLanguage(codec.getName());  
 reqOrd.setOntology(ontology.getName());  
  
  
 reqOrd.setConversationId("supplier-parts");  
 SendOrder = new SendOrder();  
 sendOrder.setCustomer(this.myAgent.getAID());  
 sendOrder.setOrder(order);  
  
  
 Action request = new Action();  
 request.setAction(sendOrder);  
 request.setActor(manufacturer.get(0));  
  
 System.*out*.println("Supplier has sent parts requested for: Order ID: " + sendOrder.getOrder());  
 try {  
 getContentManager().fillContent(reqOrd, request); //send the wrapper object  
 send(reqOrd);  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe) {  
 }  
  
 daysPassed++;  
 }  
 }  
 }  
 }  
}

6.Component delivery times are correct (supplier 2)

private class sendParts extends OneShotBehaviour {  
 public sendParts(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 if(daysPassed == 4) {  
 daysPassed = 0;  
 if (itemList != null) {  
 DFAgentDescription manufacturerTemplate = new DFAgentDescription();  
 ServiceDescription sd = new ServiceDescription();  
 sd.setType("manufacturer");  
 manufacturerTemplate.addServices(sd);  
 try {  
 DFAgentDescription[] agentsType1 = DFService.*search*(myAgent, manufacturerTemplate);  
 for (int i = 0; i < agentsType1.length; i++) {  
 manufacturer.add(agentsType1[i].getName()); // this is the AID  
  
 }  
  
 } catch (FIPAException e) {  
 e.printStackTrace();  
 }  
  
  
 ACLMessage reqOrd = new ACLMessage(ACLMessage.*REQUEST*);  
  
 reqOrd.addReceiver(manufacturer.get(0));  
 reqOrd.setLanguage(codec.getName());  
 reqOrd.setOntology(ontology.getName());  
  
  
 reqOrd.setConversationId("supplier-parts");  
 SendOrder = new SendOrder();  
 sendOrder.setCustomer(this.myAgent.getAID());  
 sendOrder.setOrder(order);  
  
  
 Action request = new Action();  
 request.setAction(sendOrder);  
 request.setActor(manufacturer.get(0));  
  
 try {  
 getContentManager().fillContent(reqOrd, request); //send the wrapper object  
 send(reqOrd);  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe) {  
 }  
  
  
 }  
 daysPassed++;  
 }  
 }   
 }  
}

7.Per-component per-day warehouse storage cost

public class WarehouseTax extends OneShotBehaviour{  
 public WarehouseTax (Agent a){  
 super(a);  
 }  
  
 @Override  
 public void action(){  
 dailyWarehouseCost = 0;  
 warehouse.entrySet().forEach(entry->{  
 totalComponents += entry.getValue();  
 });  
 dailyWarehouseCost= totalComponents \* 5;  
  
 }  
}

8.Order can only be shipped if it contains the correct parts

public void action() {  
  
 int no\_of\_4gb = 0;  
 int no\_of\_8gb = 0;  
 int no\_of\_5 = 0;  
 int no\_of\_7 = 0;  
 int no\_of\_3000 = 0;  
 int no\_of\_2000 = 0;  
 int no\_of\_64gb = 0;  
 int no\_of\_256gb = 0;  
  
 int warehouse4gb = 0;  
 int warehouse8gb = 0;  
 int warehouse5 = 0;  
 int warehouse7 = 0;  
 int warehouse3000 = 0;  
 int warehouse2000 = 0;  
 int warehouse64gb = 0;  
 int warehouse256gb = 0;  
  
 for (Map.Entry<String, Integer> entry : warehouse.entrySet()) {  
 if (entry.getKey() == "4GBRam") {  
 warehouse4gb = entry.getValue();  
 } else if (entry.getKey() == "8GBRam") {  
 warehouse8gb = entry.getValue();  
 } else if (entry.getKey() == "5Screen") {  
 warehouse5 = entry.getValue();  
 } else if (entry.getKey() == "7Screen") {  
 warehouse7 = entry.getValue();  
 } else if (entry.getKey() == "3000Battery") {  
 warehouse3000 = entry.getValue();  
 } else if (entry.getKey() == "2000Battery") {  
 warehouse2000 = entry.getValue();  
 } else if (entry.getKey() == "64GBStorage") {  
 warehouse64gb = entry.getValue();  
 } else if (entry.getKey() == "256GBStorage") {  
 warehouse256gb = entry.getValue();  
 }  
 }  
  
 for (int i = 0; i < sendToCustomer.getPhoneOrderQuantity(); i++) {  
 try {  
 if (itemList.get(i) instanceof Screen) {  
 if (((Screen) itemList.get(i)).getLength() == 5) {  
 no\_of\_5++;  
 }  
  
 } else {  
 no\_of\_7++;  
 }  
  
 if (itemList.get(i) instanceof Battery) {  
 if (((Battery) itemList.get(i)).getCapacity() == 2000) {  
 no\_of\_2000++;  
 } else {  
 no\_of\_3000++;  
 }  
 }  
 if (itemList.get(i) instanceof Storage) {  
 if (((Storage) itemList.get(i)).getSpace() == 64) {  
 no\_of\_64gb++;  
 } else {  
 no\_of\_256gb++;  
 }  
 }  
 if (itemList.get(i) instanceof Ram) {  
 if (((Ram) itemList.get(i)).getSize() == 4) {  
 no\_of\_4gb++;  
 } else {  
 no\_of\_8gb++;  
 }  
 }  
 } catch (NullPointerException ne) {  
  
 } catch (IndexOutOfBoundsException ie) {  
  
 }  
 }  
  
  
 int partsInStock = 0;  
  
  
 try {  
  
  
 if (warehouse4gb >= no\_of\_4gb) {  
 warehouse.put("4GBRam", warehouse.get("4GBRam") - no\_of\_4gb);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 4gb Ram to complete order today");  
 }  
  
 if (warehouse8gb >= no\_of\_8gb) {  
 warehouse.put("8GBRam", warehouse.get("8GBRam") - no\_of\_8gb);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 8gb Ram to complete order today");  
 }  
  
 if (warehouse256gb >= no\_of\_256gb) {  
 warehouse.put("256GBStorage", warehouse.get("256GBStorage") - no\_of\_256gb);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 256gb storage to complete order today");  
 }  
  
 if (warehouse64gb >= no\_of\_64gb) {  
 warehouse.put("64GBStorage", warehouse.get("64Storage") - no\_of\_64gb);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 64gb storage to complete order today");  
 }  
  
 if (warehouse2000 >= no\_of\_2000) {  
 warehouse.put("2000Battery", warehouse.get("2000Battery") - no\_of\_2000);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 2000 battery to complete order today");  
 }  
  
 if (warehouse3000 >= no\_of\_3000) {  
 warehouse.put("3000Battery", warehouse.get("3000Battery") - no\_of\_3000);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 3000 battery to complete order today");  
 }  
  
 if (warehouse5 >= no\_of\_5) {  
 warehouse.put("5Screen", warehouse.get("5Screen") - no\_of\_5);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 5 inch screens to complete order today");  
 partsInStock++;  
 }  
  
 if (warehouse7 >= no\_of\_7) {  
 warehouse.put("7Screen", warehouse.get("7Screen") - no\_of\_7);  
 partsInStock++;  
 } else {  
 System.*out*.println("Not enough 7 inch screens to complete order today");  
 }  
 } catch (NullPointerException ne) {  
  
 }  
  
 if (phonesMade + (int)order.getPhoneOrderQuantity() <= 50 )  
 {  
  
  
 if (partsInStock == 4) {  
 ACLMessage reqOrd = new ACLMessage(ACLMessage.*REQUEST*);  
  
 reqOrd.addReceiver(customers.get(0));  
 reqOrd.setLanguage(codec.getName());  
 reqOrd.setOntology(ontology.getName());  
  
  
 reqOrd.setConversationId("completed-order");  
 SendOrder = new SendOrder();  
 sendOrder.setCustomer(this.myAgent.getAID());  
 sendOrder.setOrder(sendToCustomer);  
  
  
 phonesMade = phonesMade + (int) order.getPhoneOrderQuantity();  
  
 Action request = new Action();  
 request.setAction(sendOrder);  
 request.setActor(customer);  
  
 try {  
 getContentManager().fillContent(reqOrd, request); //send the wrapper object  
 send(reqOrd);  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe) {  
  
 }  
 }  
  
 }  
 }  
  
}

9.Max of 50 smartphones

if (phonesMade + (int)order.getPhoneOrderQuantity() <= 50 )  
{  
  
  
if (partsInStock == 4) {  
 ACLMessage reqOrd = new ACLMessage(ACLMessage.*REQUEST*);  
  
 reqOrd.addReceiver(customers.get(0));  
 reqOrd.setLanguage(codec.getName());  
 reqOrd.setOntology(ontology.getName());  
  
  
 reqOrd.setConversationId("completed-order");  
 SendOrder = new SendOrder();  
 sendOrder.setCustomer(this.myAgent.getAID());  
 sendOrder.setOrder(sendToCustomer);  
  
  
 phonesMade = phonesMade + (int) order.getPhoneOrderQuantity();  
  
 Action request = new Action();  
 request.setAction(sendOrder);  
 request.setActor(customer);  
  
 try {  
 getContentManager().fillContent(reqOrd, request); //send the wrapper object  
 send(reqOrd);  
  
 } catch (Codec.CodecException ce) {  
 ce.printStackTrace();  
 } catch (OntologyException oe) {  
  
 }  
}

10.Profit calculation

public endDay(Agent a) {  
 super(a);  
 }  
  
 @Override  
 public void action() {  
 ACLMessage msg = new ACLMessage(ACLMessage.*INFORM*);  
 msg.addReceiver(tickerAgent);  
 suppliers.clear();  
 profit = currentMoney - dailyWarehouseCost;  
 System.*out*.println("Todays profit: £" + profit);  
 phonesMade = 0;  
 msg.setContent("done");  
 myAgent.send(msg);  
 }  
  
}

11.Order acceptance/rejection

if (order.getOrderCost() < 350) {  
 System.*out*.println("Order does not meet minimum amount of 350. Order Rejected");  
 order = new Order();