Edinburgh Napier University

SET10111 Multi-Agent Systems

Practical textbook

Dr Simon Powers

Contents

Unit 1

Introduction to Agent-Oriented Programming with JADE

This textbook is designed to supplement the practical and online resources for SET10111 Multi-Agent Systems. The goal of the practical part of the module is to give you a strong underpinning in the agent-oriented programming paradigm. Although there are many agent-oriented programming languages, we will be using JADE (the Java Agent DEvelopment framework, `https://jade.tilab.com/`). This is free open source software developed by Telecom Italia, and is widely used to develop multi-agent systems. JADE acts as middleware to add agent functionality on top of Java, and is accessed by means of a Java API. This means that you can leverage the programming skills that you have already learnt in Java or C# and take them forward to the world of agents.

1.1 What is agent-oriented programming?

In the lectures we introduced the idea of an intelligent *agent*. An intelligent acts autonomously (i.e. without continual direction from its owner) to achieve the goals set by its owner. This is challenging because the environment that it acts in may change (e.g. new obstacles may appear in the path of a robot), and the agent may have to interact with other agents that represent the interests of other users. Moreover, the interests of the users may not be the same (and may even be opposing). Consider, for example, a software agent that attempts to create a package holiday for its owner at the best possible price, while satisfying its owner's preferences for a beach holiday on a tropical island with easy access to discos. This agent will need to be able to negotiate with other agents representing the interests of airlines, hotels, beach resorts etc, who will be trying to maximise the amount of money they receive for their owners. Our agent therefore needs to be able to:

- find the IP addresses of agents representing hotels, airlines...;
- send messages to other agents, e.g. to enquire the price of something;
- have a shared understanding with the other agents of what concepts like a hotel room and a flight are (i.e. the agents need to speak the same language);
- react to offers received (e.g. decide whether to accept a flight to Ibiza for 200 GBP at 6am on Monday morning or not).

We are going to develop the skills to do all of these things in this textbook. We will start with agents that have very simple goals, e.g. print a hello world message every 10 seconds, and move on to agents that need to ask for the help of other agents to achieve their goals, and finally to agents that can fully communicate with each other using a common language (ontology).

1.1.1 Message passing between agents

Note that we just mentioned agents *asking for the help of other agents*. This is not a polite choice of words but is actually fundamental to the agent-oriented programming paradigm. You will all be familiar with object-oriented programming. In object-oriented programming, one object simply uses another object by calling one of its public methods. The other object has no say in the matter – if one of its public methods is called, it has to execute the code within that method. But this is not appropriate at all when considering the scenario above – our agent should not be able to force a hotel agent to sell us a room at any price we like simply by calling its sell method!!! In general, one agent cannot force another agent to do something, as that other agent may have different goals (and a different owner). An agent has control over both its internal state (as in object-oriented programming) and over its actions (unlike object-oriented programming).

What does all this mean? **It means that one agent does not call a method on another agent**. Instead, Agent x sends a message to Agent y, and Agent y then chooses how to handle this message. Much of agent-oriented programming therefore consists of writing code – *behaviours* – to send outgoing messages and react to incoming messages. **As a general rule, agents should never call methods on each other**. Although agents are represented as objects in JADE, one agent should **not** have a reference variable pointing to the object behind another agent. Instead, the agent should only know the Agent Identifiers (AIDs) of the other agents in the system, and should interact with other agents by sending messages addressed to these AIDs. An agent should only call methods on itself or on other non-agent Java classes (e.g. from the standard Java API).

1.1.2 Agent behaviours

An agent in JADE consists of a number of concurrently executing *behaviours*. For example, an agent representing a hotel might have one behaviour listening for messages containing queries about the price of rooms, and another behaviour listening for messages making reservations. In general, if you want your agent to do something, the code for that has to be placed inside a behaviour. Behaviours are implemented as subclasses of the JADE API class `Behaviour`.

1.2 Getting started with JADE

Ok, it's time to have a go at creating our first agents! In our examples, we will be working with the Eclipse IDE. However, this is not prescriptive and you may use any IDE that you prefer. To get going with JADE in Eclipse, follow these steps:

- Download the latest version of JADE from `https://jade.tilab.com/download/jade/`. Download JadeAll and unzip it in a suitable folder, then unzip the 4

zip archives within it . You may wish to download these files to a location such as your h: drive so they are available for future use.

- Start Eclipse.
- Create a new workspace and a new project.
- To allow your project work with Eclipse go into project properties, then select Java Build Path, then add jade.jar (it's in the jade-bin directory) as an external JAR.

The simplest JADE agent that we can create looks like this:

```
1 import jade.core.Agent;
2 public class SimpleAgent extends Agent {
3
4   //This method is called when the agent is launched
5   protected void setup() {
6     // Print out a welcome message
7     System.out.println("Hello! Agent "+getAID().getName()+" is
          ready.");
8   }
9 }
```

Listing 1.1: The world's simplest agent

All agents are implemented as subclasses of the `Agent` superclass. The `setup()` method runs when the agent is created (you can think of it as like a constructor in object-oriented programming). At the moment all this agent will do is run the code in its setup method, which prints out a hello message. It doesn't have any behaviours yet, so can't do anything interesting.

Note that we can get the agent's unique identifier, its AID, by calling its `getAID()` method. Every agent has a unique AID. We're going to see how to run our agent and set its AID now.

1.2.1 Launching agents

To launch an agent, we first need start up the JADE platform, which will create a container for our use and create an RMA agent which allows our container to be administered via the GUI.

The JADE platform is configured and started from the `main` method of your Java application as follows:

```
1 Profile myProfile = new ProfileImpl();
2 Runtime myRuntime = Runtime.instance();
3 ContainerController myContainer = myRuntime.createMainContainer(
      myProfile);
4 AgentController rma = myContainer.createNewAgent("rma", "jade.tools
      .rma.rma", null);
5 rma.start();
```

Listing 1.2: Starting the JADE platform from with the `main` method

The above code will lunch JADE from within a Java program and create a container (accessed via the object `myContainer`). The last two lines show how we launch an agent, in this case, the Remote Monitoring Agent that provides a GUI showing all agents in the system. Note the name of the agent is "rm" and the actual

class containing the agent is called `jade.tools.rma.rma` . The `AgentController` object represents the agent and the `start()` method launches our new agent. Our `SimpleAgent` would similarly be launched with the lines:

```
1 AgentController myAgent = myContainer.createNewAgent("Fred",
     SimpleAgent.class.getCanonicalName() , null);
2 myAgent.start();
```

Listing 1.3: Launching our `SimpleAgent` from the application's `main` method

Here we are telling JADE to make a new agent according to the definition in the `SimpleAgent` class, and to give the agent the local name Fred. This local name forms the first part of the agent's AID (the second part is the IP address of the machine on which the JADE container is running), and will be returned by calling `getAID().getLocalName()` on the agent. It's therefore wise to give your agent's meaningful names when you create them!

The complete `main` method to start the JADE platform and then launch a SimpleAgent called Fred is therefore:

```
1  import jade.core.Profile;
2  import jade.core.ProfileImpl;
3  import jade.wrapper.AgentController;
4  import jade.wrapper.ContainerController;
5  import jade.core.Runtime;
6
7  public class Application {
8
9  public static void main(String[] args){
10   //Setup the JADE environment
11   Profile myProfile = new ProfileImpl();
12   Runtime myRuntime = Runtime.instance();
13   ContainerController myContainer = myRuntime.createMainContainer(
        myProfile);
14   try{
15   //Start the agent controller, which is itself an agent (rma)
16     AgentController rma = myContainer.createNewAgent("rma", "jade.
          tools.rma.rma",     null);
17     rma.start();
18
19   //Now start our own SimpleAgent, called Fred.
20     AgentController myAgent = myContainer.createNewAgent("Fred",
             SimpleAgent.class.getCanonicalName(), null);
21     myAgent.start();
22
23     }catch(Exception e){
24       System.out.println("Exception starting agent: " + e.toString
            ());
25     }
26   }
27 }
```

Listing 1.4: Starting the JADE platform from with the `main` method

If you compile and run your application, the JADE platform will start, Fred will be created, and the hello method from Fred will be printed to the console. You should also see the JADE GUI provided by the Remote Management Agent. This will show all of the agents currently in the system, and should look like that in Figure 1.1.

Note that the other agents are the Agent Management System, the Directory Facilitator, and the Remote Management Agent. The Directory Facilitator (DF)

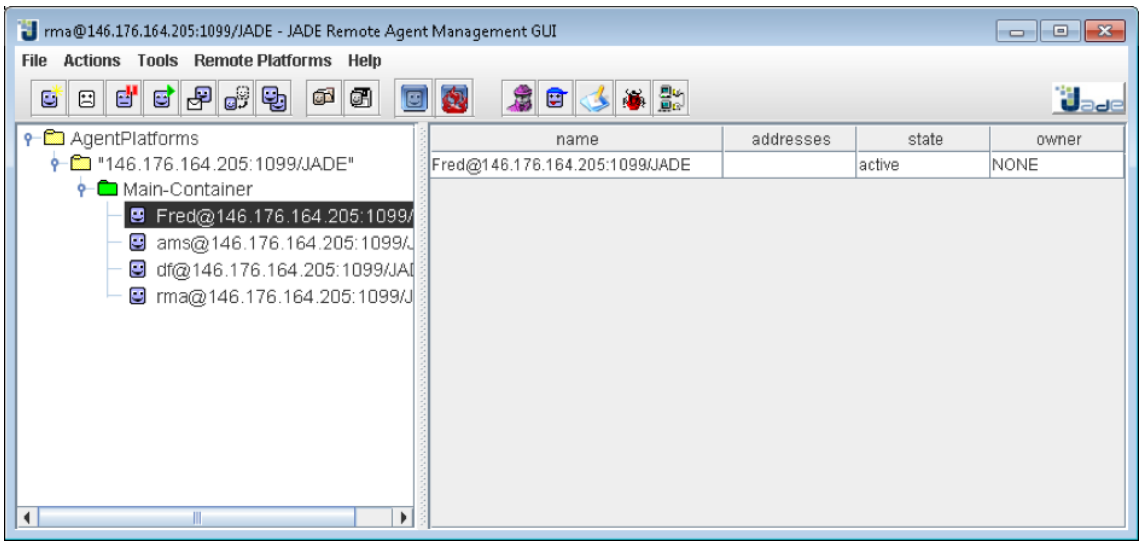

Figure 1.1: The JADE GUI

agent allows us to register our agent in a "yellow pages" directory so that its address (AID) can be found by other agents who may wish to send it messages.

1.2.2 Registering and deregistering our agents with the yellow pages directory

We can register our agent with the DF by placing the following code in its `setup()` method.

```
1  DFAgentDescription dfd = new DFAgentDescription();
2  dfd.setName(getAID());
3  ServiceDescription sd = new ServiceDescription();
4  sd.setType("Simple-agent");
5  sd.setName(getLocalName() + "-Simple-agent");
6  dfd.addServices(sd);
7  try{
8    DFService.register(this, dfd);
9  }
10 catch(FIPAException e){
11   e.printStackTrace();
12 }
```

Listing 1.5: Registering your agent with the yellow pages directory (place this in the `setup()` method

You will need to import the relevant classes, i.e. `DFAgentDescription`, `ServiceDescription`, `DFService` and `FIPAException`. This code is registering our agent's AID (line 2) along with a description of the service that the agent provides to others (in this case "Simple agent"). You should get into the habit of registering all of your agents with the DF so that they can be found at runtime by other agents (possibly running on other computers and written by other people).

When your agent terminates, it should deregister with the DF so that other agents don't try to contact a dead agent! Every agent automatically executes its `takeDown()` method when it terminates. To deregister, we should place the following code in our `takeDown()` method.

```
1  protected void takeDown(){
2    //Deregister from the yellow pages
```

```
3 try{
4   DFService.deregister(this);
5 }
6 catch(FIPAException e){
7   e.printStackTrace();
8 }
```

Listing 1.6: Deregistering your agent with the yellow pages directory when it terminates

As you can see, deregistering is very easy (so no excuse not to do it!!). To kill an agent, you call its `doDelete()` method, which automatically calls the `takeDown()` method before killing the agent. Alternatively, you can kill an agent by selecting it in the JADE GUI.

1.3 Behaviours

At the moment our agent doesn't really do anything, other than print a hello world message when it is created. In order for our agents to do anything, they need to have *behaviours*. A behaviour allows us to specify how an agent will respond to a particular event (e.g. receiving a message from another agent). In object-oriented programming you will be used to thinking of objects as passive entities that only do things when the appropriate methods are called on them by another object. By contrast, agents are *autonomous* – each agent has its own thread of control, which it uses to control its lifecycle and decide when to perform which actions.

Crucially, an agent can execute several behaviours concurrently. Behaviours are implemented as subclasses of the `jade.core.behaviours.Behaviour` class. Behaviour execution is cooperative (not pre-emptive). This means that you as the programmer decide when your agent will switch between behaviours.

The agent below creates a `TickerBehaviour`. `TickerBehaviours` call an `onTick()` event at a specified interval (in this case every 1000 ms). This is a way to get your agent to keep the performing the same action every t milliseconds.

```
1  import jade.core.Agent;
2  import jade.core.behaviours.TickerBehaviour;
3
4  public class TimerAgent extends Agent {
5    int w = 15;
6    public void setup() {
7      //Create a new TickerBehaviour
8      addBehaviour(new TickerBehaviour(this, 1000) {
9        //call onTick every 1000ms
10
11       protected void onTick() {
12         //Count down
13         if (w > 0) {
14           System.out.println(w + " seconds left.");
15           w--;
16         } else {
17           System.out.println("Bye, bye");
18           myAgent.doDelete(); //Delete this agent
19
20         }
21       }
22     });
23   }
```

```
24 }
```

Listing 1.7: A timer agent

We add the behaviour by calling the agent's `addBehaviour()` method. This is done in the `setup()` method. In general, your `setup()` method must add at least one behaviour before it terminates, otherwise your agent will not do anything after the `setup()` method terminates.

Note that we have created a new instance of the library `TickerBehaviour` class, and overridden its `onTick()` method. This behaviour simply counts down from 15 to 1 every second, and then kills the agent. Every behaviour has an instance variable called `myAgent` that refers to the agent that is executing the behaviour, and which you set through the behaviour's constructor. Although our example behaviour is simple, `TickerBehaviours` are very commonly used, because you often do want your agent to keep doing a certain action every t milliseconds. For example, an agent that is searching for a particular book to buy on the internet may want to periodically check for any new sellers. A `TickerBehaviour` is ideal for this. The following code searches for any new `SimpleAgents` once every 60 seconds:

```
1  private ArrayList<AID> simpleAgents = new ArrayList<>();
2
3  protected void setup(){
4    ...
5    addBehaviour(new TickerBehaviour(this, 60000){
6      protected void onTick(){
7        //create a template for the agent service we are looking for
8        DFAgentDescription template = new DFAgentDescription();
9        ServiceDescription sd = new ServiceDescription();
10       sd.setType("Simple-agent");
11       template.addServices(sd);
12       //query the DF agent
13       try{
14         DFAgentDescription[] result = DFService.search(myAgent,
               template);
15         simpleAgents.clear(); //we're going to replace this
16         for(int i=0; i<result.length; i++){
17           simpleAgents.add(result[i].getName()); // this is the AID
18         }
19       }
20       catch(FIPAException e){
21         e.printStackTrace();
22       }
23     }
24   }
25 }
```

Listing 1.8: A ticker behaviour to search for new simple agents every 60 seconds

1.3.1 Switching behaviours

As we mentioned above, behaviour switching in JADE is *cooperative*. The two fundamental methods that a behaviour has are `public void action()` and `public boolean done()`. When a behaviour is executed its `action()` method runs until it returns. While the `action()` method is running no other behaviours of the agent will be running – all of the agent's other behaviours will be waiting in that agent's behaviour queue. Therefore by deciding when the `action()` method returns you

decide when the behaviour will stop running and the next behaviour will start. This is fundamental to the way that JADE handles concurrent behaviour scheduling. Each behaviour in the agent's behaviour queue runs in turn. The queue then wraps around to the start and the first behaviour executes again.

Behaviours will remain in the agent's behaviour queue for wrap-around execution until you remove them from the queue. A behaviour is removed from the queue when its `done()` method returns `true`. More precisely, when a behaviour's `action()` method returns then the agent's scheduler calls `done()` on that behaviour and removes the behaviour from the queue if the call to `done()` returns `true`.

As an example of concurrent execution of behaviours, consider the class `DemoAgent`. This agent has two behaviours, `B1` and `B2`. You should run this agent to demonstrate how execution of the two behaviours proceeds. Note how `B1` executes until its `action()` method returns, then `B2` executes, then `B1` again and so on. `B1` then gets removed from the behaviour queue after 10 executions, since this is when its `done()` method first returns `true`. If we change the code to add behaviour `B2` first in the `setup()` method then `B2` will execute first followed by `B1`. Note that the constructor of both behaviours takes in a reference to the agent that will be executing them – this gets stored in the instance variable `myAgent`, which is inherited from the `Behaviour` superclass. This allows us to, for example, print out the agent's name on line 15. Finally, note that `B1` and `B2` are inner classes of `DemoAgent`. This does not have to be the case, you could make them as normal classes in their own `.java` files. However, making them as inner classes allows them to access the private features of `DemoAgent`, if need be. This is a common programming style in JADE.

```
1  public class DemoAgent extends Agent {
2
3    protected void setup() {
4      addBehaviour(new B1(this));
5      addBehaviour(new B2(this));
6    }
7
8    public class B1 extends Behaviour {
9      private int timesCalled = 0;
10     public B1(Agent a){
11       myAgent = a;
12     }
13     @Override
14     public void action() {
15       System.out.println(myAgent.getLocalName());
16       timesCalled++;
17
18     }
19
20     @Override
21     public boolean done() {
22       return timesCalled >= 10;
23     }
24
25   }
26
27   public class B2 extends Behaviour {
28     private int timesCalled = 0;
29     public B2(Agent a){
30       myAgent = a;
31     }
32     @Override
```

```
33      public void action() {
34        System.out.println(timesCalled);
35        timesCalled++;
36
37      }
38
39      @Override
40      public boolean done() {
41        return timesCalled >= 20;
42      }
43
44    }
45  }
```

Listing 1.9: An agent that demonstrates concurrent execution of behaviours.

Note that we should not perform very long computations in one go in the `action()` method, as no other behaviours will be able to run until it returns. Long computations should be split up into a series of smaller steps, with the state (intermediate results) of the computation saved in instance variables of the behaviour or agent. You certainly need to avoid having endless loops in the `action()` method of a behaviour, otherwise no other behaviours will ever run!

The following code illustrates how to break the action method up into separate stages, saving the state of the computation after each stage. The `onEnd()` method is inherited from the `Behaviour` superclass and executes as the behaviour is removed from the agent's behaviour queue, i.e. when `done()` returns true. It is used to perform any clean up actions that are needed, e.g. freeing resources. It returns an `int` giving the error status of the behaviour (e.g. 0 for normal operation and other numbers for error codes).

```
1  public class ThreeStepBehaviour extends Behaviour {
2    private int step = 1;
3    private int result = 0;
4    private boolean finished = false;
5
6    public ThreeStepBehaviour(Agent a){
7      //call the parent constructor,
8      //to set the myAgent reference
9      super(a);
10   }
11
12   @Override
13   public void action() {
14     switch(step){
15       case 1:
16         result += 10;
17         System.out.println(result);
18         step++;
19         break;
20       case 2:
21         result += 100;
22         System.out.println(result);
23         step++;
24         break;
25       case 3:
26         result += 200;
27         System.out.println(result);
28         finished = true;
29         break;
```

```
30
31      }
32
33    }
34
35    @Override
36    public boolean done() {
37      return finished;
38    }
39
40    @Override
41    public int onEnd() {
42      System.out.println("Terminating behaviour");
43      return 0;
44    }
45  }
```

Listing 1.10: A multistep behaviour that saves its state during execution.

If the code in the case blocks was more complex then we could separate it out into private helper methods in the Behaviour class. Note that a behaviour like this that saves its complete state in its instance variables is essentially a finite state machine.

Multistep behaviours like this are very common in JADE. For example, a behaviour to buy a book might be broken down into three steps: 1. Send a message to seller agents to enquire whether they have a book and its price; 2. Collate the responses from sellers and find the cheapest; 3. Send an order to the cheapest seller. We need to break this up into separate steps so that we do not, for example, block the agent from doing anything else while it is waiting for a response from each seller. We will cover this in more detail in the next chapter.

In fact, because multistep behaviours are so common, JADE provides a class called `SequentialBehaviour` that can be used to implement them without needing to use a `switch` block. Instead, each step of the behaviour is written in its own class, and these behaviours are then added as sub-behaviours of the sequential behaviour. The following agent demonstrates this.

```
1  public class SequentialAgent extends Agent {
2    private int result = 0;
3
4    @Override
5    protected void setup() {
6      SequentialBehaviour s1 = new SequentialBehaviour(this);
7      s1.addSubBehaviour(new B1());
8      s1.addSubBehaviour(new B2());
9      s1.addSubBehaviour(new B3());
10     addBehaviour(s1);
11   }
12
13   public class B1 extends OneShotBehaviour {
14
15     @Override
16     public void action() {
17       result += 10;
18       System.out.println(result);
19     }
20
21   }
22
23   public class B2 extends OneShotBehaviour {
24
```

```
25      @Override
26      public void action() {
27        result += 100;
28        System.out.println(result);
29      }
30
31    }
32
33    public class B3 extends OneShotBehaviour {
34
35      @Override
36      public void action() {
37        result += 200;
38        System.out.println(result);
39      }
40
41    }
42  }
```

Listing 1.11: A multistep behaviour that saves its state during execution.

A `SequentialBehaviour` executes it first sub-behaviour until that behaviour's `done()` method returns `true`, and then executes its next sub-behaviour. The `done()` method of a `OneShotBehaviour` always returns `true`.

Sometimes you might want a behaviour to keep executing repeatedly in a cyclic manner. To do this, we can modify our `onEnd()` method so that it adds the behaviour back into our agent's behaviour queue. We also need to override the `reset()` method so that it resets the state of the behaviour back to its initial state when the behaviour was initialised. Reseting the state allows us to use the same Behaviour object again, avoiding the overhead of creating a new object and subsequent garbage collection of the old object.

```
1  public class ThreeStepBehaviour extends Behaviour {
2
3    @Override
4    public int onEnd() {
5      System.out.println("Terminating behaviour");
6      reset(); //remember to call this before adding the behaviour
           again
7      myAgent.addBehaviour(this); //add the behaviour back to the
           agent's behaviour queue
8      return 0;
9    }
10
11   //reset the behaviour's state
12   @Override
13   public void reset() {
14     step = 1;
15     result = 0;
16     finished = false;
17   }
18
19 }
```

Listing 1.12: A multistep behaviour that saves its state during execution.

1.3.2 More complex behaviours

JADE has a number of other behaviour types that extend the basic `Behaviour` superclass. One of these is `CyclicBehaviour`. This is a behaviour that keeps executing forever. A `CyclicBehaviour` overrides the `done()` method to always return false (and this overriding is final, so the cycling cannot be stopped by a subclass). Using `CyclicBehaviour` provides a simpler way to implement the previous behaviour.

```
1  public class ThreeStepBehaviour extends CyclicBehaviour {
2    int step = 1;
3    int result = 0;
4    //no need for the finished variable anymore
5
6    @Override
7    public void action() {
8      switch(step){
9        case 1:
10         result += 10;
11         System.out.println(result);
12         step++;
13         break;
14       case 2:
15         result += 100;
16         System.out.println(result);
17         step++;
18         break;
19       case 3:
20         result += 200;
21         System.out.println(result);
22         reset(); //now we call reset here after the final step
23         break;
24
25     }
26
27   }
28
29   @Override
30   public void reset() {
31     step = 1;
32     result = 0;
33   }
34 }
```

Listing 1.13: A multistep behaviour that saves its state during execution.

Note that we no longer have to implement the `done()` method. Cyclic behaviours are typically used when your agent needs to act as a server to handle requests from other agents at any time. For example, an agent selling books would have a cyclic behaviour constantly executing to receive enquiries from buyer agents. Ticker behaviours, implemented in the `TickerBehaviour` class are related. They are cyclic behaviours that should repeat every t milliseconds.

Another commonly used behaviour is the one-shot behaviour, implemented by the class `OneShotBehaviour`. This is a behaviour whose `action()` method executes exactly once, after which the behaviour is removed from the agent's behaviour queue. It does this by overriding the `done()` method to always return true. It is therefore the opposite of the `CyclicBehaviour` class. Waker behaviours are one-shot behaviours that execute after a specified period of time, passed into their constructor. There are also Parallel Behaviours and Finite State Machine behaviours in the JADE API,

which you can find out more about in the JADE documentation.

Unit 2

Communication between agents

So far in this textbook we've only been dealing with applications that have a single agent. But where things gets really interesting is when we have multiple agents running at the same time (which may or may not have the same owner). For example, a travel booking system will have agents representing customers, airlines, hotels, etc. Clearly, to do anything useful these agents will need to be able to communicate with each other.

Communication is handled through message passing between agents. JADE implements this for us, in a way that is compliant with the FIPA (Foundation for Intelligent Physical Agents) standard for cross-platform agent communication. In particular, JADE implements FIPA's Agent Communication Language (ACL).

2.1 Sending and receiving a basic message

To demonstrate message passing, let's consider a simple application with two types of agents: agents that send messages and agents that receive messages from them and print that message to the console. First, let's look at how we send messages. The agent below uses a `TickerBehaviour` to send a message to all receiver agents every ten seconds.

```
1  import java.util.ArrayList;
2  import jade.core.AID;
3  import jade.core.Agent;
4  import jade.core.behaviours.TickerBehaviour;
5  import jade.domain.DFService;
6  import jade.domain.FIPAException;
7  import jade.domain.FIPAAgentManagement.DFAgentDescription;
8  import jade.domain.FIPAAgentManagement.ServiceDescription;
9  import jade.lang.acl.ACLMessage;
10
11 public class SenderAgent extends Agent {
12
13   private ArrayList<AID> receiverAgents = new ArrayList<>();
14
15   @Override
16   protected void setup() {
17     //add this agent to the yellow pages
18     DFAgentDescription dfd = new DFAgentDescription();
19     dfd.setName(getAID());
20     ServiceDescription sd = new ServiceDescription();
21     sd.setType("sender-agent");
22     sd.setName(getLocalName() + "-sender-agent");
```

```
23      dfd.addServices(sd);
24      try{
25        DFService.register(this, dfd);
26      }
27      catch(FIPAException e){
28        e.printStackTrace();
29      }
30
31      //add behaviour to find new receiver agents
32      addBehaviour(new SearchYellowPages(this,10000));
33      //add the behaviour to send a message to each receiver
34      //every ten seconds
35      addBehaviour(new SenderBehaviour(this,10000));
36
37    }
38
39    protected void takeDown(){
40      //Deregister from the yellow pages
41      try{
42        DFService.deregister(this);
43      }
44      catch(FIPAException e){
45        e.printStackTrace();
46      }
47    }
48
49    public class SearchYellowPages extends TickerBehaviour {
50
51      public SearchYellowPages(Agent a, long period) {
52        super(a, period);
53      }
54
55      @Override
56      protected void onTick() {
57        //create a template for the agent service we are looking for
58        DFAgentDescription template = new DFAgentDescription();
59        ServiceDescription sd = new ServiceDescription();
60        sd.setType("receiver-agent");
61        template.addServices(sd);
62        //query the DF agent
63        try{
64          DFAgentDescription[] result = DFService.search(myAgent,
              template);
65          receiverAgents.clear(); //we're going to replace this
66          for(int i=0; i<result.length; i++){
67            receiverAgents.add(result[i].getName()); // this is the
                AID
68          }
69        }
70        catch(FIPAException e){
71          e.printStackTrace();
72        }
73
74      }
75
76    }
77
78    public class SenderBehaviour extends TickerBehaviour {
79
80      public SenderBehaviour(Agent a, long period) {
```

```
81        super(a, period);
82      }
83
84      @Override
85      protected void onTick() {
86        //send a message to all receiver agents
87        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
88        msg.setContent("hello from agent " + myAgent.getLocalName());
89        //add receivers
90        for(AID receiver : receiverAgents) {
91          msg.addReceiver(receiver);
92        }
93        myAgent.send(msg);
94      }
95
96    }
97
98  }
```

Listing 2.1: The sender agent

The main behaviour we are interested in here is the `SenderBehaviour`, and in particular the contents of its `onTick()` method. This sends a message to all of the receiver agents with the message "hello from agent " followed by the agent's name. To send a message, we create an `ACLMessage` object. The constructor of an `ACLMessage` takes in the *performative* of the message. The performative of the message is its type (more formally, its action, i.e. the effect that it performs on the beliefs of the receiving agent; you will learn more about this in the lecture series). The performative of the message we are sending is `INFORM`, i.e. the message provides the receiving agent with a piece of information about the world. We set the list of receiving agents' AIDs using the `addReceiver(AID id)` method. A message can be sent to one or more recipients at the same time through repeated calls to `addReceiver`. To send the message, we simply invoke the agent's `send(ACLMessage msg)` method, which we access in the behaviour through the behaviour's `myAgent` instance variable (remember that the reference to the agent executing the behaviour is passed to the behaviour in its constructor).

To find receiving agents, we use another `TickerBehaviour` to query the directory facilitator agent every ten seconds to find the AIDs of all receiving agents. We store these in an `ArrayList` inside the agent that the behaviour is running on (for convenience, we make the behaviour an inner class of this agent). We also register ourselves with the directory facilitator in the `setup()` method. Note also that we are good and deregister ourselves in the `takeDown()` method.

Our receiving agent use a `CyclicBehaviour` to process incoming messages as they arrive, as shown in the code below.

```
1  import jade.core.Agent;
2  import jade.core.behaviours.CyclicBehaviour;
3  import jade.core.behaviours.TickerBehaviour;
4  import jade.domain.DFService;
5  import jade.domain.FIPAException;
6  import jade.domain.FIPAAgentManagement.DFAgentDescription;
7  import jade.domain.FIPAAgentManagement.ServiceDescription;
8  import jade.lang.acl.ACLMessage;
9
10 public class ReceiverAgent extends Agent {
11
12   @Override
```

```
13   protected void setup() {
14     //add this agent to the yellow pages
15     DFAgentDescription dfd = new DFAgentDescription();
16     dfd.setName(getAID());
17     ServiceDescription sd = new ServiceDescription();
18     sd.setType("receiver-agent");
19     sd.setName(getLocalName() + "-receiver-agent");
20     dfd.addServices(sd);
21     try{
22       DFService.register(this, dfd);
23     }
24     catch(FIPAException e){
25       e.printStackTrace();
26     }
27     //add the receiver behaviour
28     addBehaviour(new ReceiverBehaviour(this));
29   }
30
31   protected void takeDown(){
32     //Deregister from the yellow pages
33     try{
34       DFService.deregister(this);
35     }
36     catch(FIPAException e){
37       e.printStackTrace();
38     }
39   }
40
41   public class ReceiverBehaviour extends CyclicBehaviour {
42
43     public ReceiverBehaviour(Agent a) {
44       super(a);
45     }
46
47     @Override
48     public void action() {
49       //try to receive a message
50       ACLMessage msg = myAgent.receive();
51       if(msg != null) {
52         //process the message
53         System.out.println("I am " + myAgent.getLocalName());
54         System.out.println("Message received from " +
55         msg.getSender());
56         System.out.println("The message is: ");
57         System.out.println(msg.getContent());
58         System.out.println();
59       }
60       else {
61         //put the behaviour to sleep until
62         //a message arrives
63         block();
64       }
65     }
66
67   }
68
69 }
```

Listing 2.2: The receiver agent

First, and importantly, note that `ReceiverBehaviour` is a cyclic behaviour. This is

because we need to continually listen for messages. Because a multi-agent system is asynchronous, we do not know when the message we are waiting for will arrive. As a general rule a one shot behaviour would not be suitable for receiving messages, as this would rely on the message already having been received when the behaviour executes for its one and only time. **In general, cyclic behaviours should be used to listen for and process messages. This is a general pattern in JADE**.

Recall also the way that behaviours are queued inside an agent. The agent will execute the `action()` method of a behaviour until the method returns. The `done()` method of the behaviour is then executed. If this returns `true` then the behaviour is removed from the agent's behaviour queue and will never be executed again. On the other hand, if the `done()` method returns `false` then the behaviour is placed in the back of the queue and the `action()` method of the next behaviour in the queue executes. A one shot behaviour has a `done()` method that always returns `true`, so the behaviour is removed after the first time it is executed. A cyclic behaviour has a `done()` method that always returns `false`, so the behaviour will always stay in the agent's behaviour queue. However, it will move to the back of the queue and so all of the other behaviours in the queue will each execute once before the cyclic behaviour executes again.

The key code for receiving a message is in the `action()` method of `ReceiverBehaviour`. When a message is sent to an agent, the receiving agent automatically adds it to its message queue (JADE takes care of this for us). Note that this is a queue in the data structure sense, i.e. messages are pulled out in a first-in first-out manner. To retrieve the message at the head of the queue, you call the agent's `receive()` method, which returns the `ACLMessage` object representing the message. If there is no message currently in the agent's queue then this returns null.

Lines 51-64 demonstrate the fundamental way that you should handle receiving messages in JADE. If there is currently no message in the queue, i.e. if the call to `receive()` returns `null`, then you should put the behaviour to sleep until a message is received. To put the behaviour to sleep, we call the `block()` method of the `Behaviour` class. This works as follows. As soon as the `action()` method returns, it moves the behaviour to the agent's blocked behaviour queue, so that it will not be scheduled anymore for execution. The behaviour becomes unblocked, and moved back to the agent's normal behaviour queue, as soon as a message is received (JADE handles this for us automatically). Note that only the behaviour that `block()` was called within is put to sleep – the other behaviours in the agent's behaviour queue continue executing normally. In other words, the whole agent is not put to sleep, only that particular behaviour (if you do actually want to put the whole agent to sleep, you can use the `blockingReceive()` method of `Agent`). The reason for putting the behaviour to sleep until a message arrives is to prevent wasting CPU cycles by keep executing a (often cyclic) behaviour that cannot proceed until a message is received. The `block` method optionally takes an argument, which is the maximum number of milliseconds to remain blocked for.

2.2 Using messages to synchronise agent behaviour

In an agent-based application you typically need your agents to syncronise their behaviour at some point of time, even though agent interactions are asynchronous. For example, you may have an agent-based automated manufacturing supply chain, where each day an agent representing a manufacturer decides which orders to accept

from customers, and which components to order from which suppliers. Similarly, each day supplier agents have to make shipments to the manufacturer.Therefore, both manufacturer and supplier agents need to know when it is the start of a new day.

To handle this synchronisation, we can create a special type of agent known as a Ticker Agent. The Ticker Agent sends a message to every other agent indicating that it is the start of a new day. When an agent has completed all of its activities for that day, it sends a message back to the Ticker Agent indicating that it is done. When the Ticker Agent has received a "done" message back from every agent it can start a new day by sending a "new day" message out to every agent again. When an agent receives a "new day" message it then begins its tasks again. The code to implement a Ticker Agent is shown below.

```
1  import java.util.ArrayList;
2
3  import jade.core.AID;
4  import jade.core.Agent;
5  import jade.core.behaviours.Behaviour;
6  import jade.domain.DFService;
7  import jade.domain.FIPAException;
8  import jade.domain.FIPAAgentManagement.DFAgentDescription;
9  import jade.domain.FIPAAgentManagement.ServiceDescription;
10 import jade.lang.acl.ACLMessage;
11 import jade.lang.acl.MessageTemplate;
12
13 public class TickerAgent extends Agent {
14
15   @Override
16   protected void setup() {
17     //add this agent to the yellow pages
18     DFAgentDescription dfd = new DFAgentDescription();
19     dfd.setName(getAID());
20     ServiceDescription sd = new ServiceDescription();
21     sd.setType("ticker-agent");
22     sd.setName(getLocalName() + "-ticker-agent");
23     dfd.addServices(sd);
24     try{
25       DFService.register(this, dfd);
26     }
27     catch(FIPAException e){
28       e.printStackTrace();
29     }
30
31     //wait for the other agents to start
32     doWait(10000);
33     addBehaviour(new SynchAgentsBehaviour(this));
34   }
35
36   @Override
37   protected void takeDown() {
38     //Deregister from the yellow pages
39     try{
40       DFService.deregister(this);
41     }
42     catch(FIPAException e){
43       e.printStackTrace();
44     }
45
46   }
```

```
47
48  public class SynchAgentsBehaviour extends Behaviour {
49
50    private int step = 0; //where we are in the behaviour
51    private int numFinReceived = 0; //number of finished messages
        from other agents
52    private ArrayList<AID> simulationAgents = new ArrayList<>();
53
54    public SynchAgentsBehaviour(Agent a) {
55      super(a);
56    }
57
58    @Override
59    public void action() {
60      switch(step) {
61      case 0:
62        //find all agents using directory service
63        //here we have two types of agent
64        //"simulation-agent" and "simulation-agent 2"
65        DFAgentDescription template1 = new DFAgentDescription();
66        ServiceDescription sd = new ServiceDescription();
67        sd.setType("simulation-agent");
68        template1.addServices(sd);
69        DFAgentDescription template2 = new DFAgentDescription();
70        ServiceDescription sd2 = new ServiceDescription();
71        sd2.setType("simulation-agent2");
72        template2.addServices(sd2);
73        try{
74          simulationAgents.clear();
75          //search for agents of type "simulation-agent"
76          DFAgentDescription[] agentsType1  = DFService.search(
              myAgent,template1);
77          for(int i=0; i<agentsType1.length; i++){
78            simulationAgents.add(agentsType1[i].getName()); // this
                is the AID
79            System.out.println(agentsType1[i].getName());
80          }
81          //search for agents of type "simulation-agent2"
82          DFAgentDescription[] agentsType2  = DFService.search(
              myAgent,template2);
83          for(int i=0; i<agentsType2.length; i++){
84            simulationAgents.add(agentsType2[i].getName()); // this
                is the AID
85            System.out.println(agentsType2[i].getName());
86          }
87        }
88        catch(FIPAException e) {
89          e.printStackTrace();
90        }
91        //send new day message to each agent
92        ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
93        tick.setContent("new day"); //the message content
94        for(AID id : simulationAgents) {
95          tick.addReceiver(id);
96        }
97        myAgent.send(tick);
98        step++;
99        break;
100     case 1:
101       //wait to receive a "done" message from all agents
```

```
102         MessageTemplate mt = MessageTemplate.MatchContent("done");
103         ACLMessage msg = myAgent.receive(mt);
104         if(msg != null) {
105           numFinReceived++;
106           if(numFinReceived >= simulationAgents.size()) {
107             step++;
108           }
109         }
110         else {
111           block();
112         }
113
114       }
115     }
116
117     @Override
118     public boolean done() {
119       return step == 2;
120     }
121
122
123     @Override
124     public void reset() {
125       step = 0;
126       numFinReceived = 0;
127     }
128
129
130     @Override
131     public int onEnd() {
132       System.out.println("End of day");
133       reset();
134       myAgent.addBehaviour(this);
135       return 0;
136     }
137
138   }
139
140 }
```

Listing 2.3: The Ticker Agent

Our Ticker Agent consists of a single behaviour. This finds all other agents in the system (lines 65-90), and then sends a "new day" message out to them (lines 92-99). It then waits to receive a "done" message back from all of the agents that it sent the "new day" message to (lines 100-112). It does this by simply counting the number of "done" messages that it has received so far, storing this in the integer instance variable of the behaviour called `numFinReceived`. When a "done" message has been received from each other agent, the behaviour resets itself and adds itself back into the Ticker Agent's behaviour queue (`done()` returns `true`, and `onEnd()` then calls the behaviour's `reset()` method).

Note that before launching its behaviour, the Ticker Agent waits for ten seconds for other agents in the system to start (line 32). It does this by calling the `doWait()` method inherited from `Agent`, which puts the agent to sleep for a given number of milliseconds.

As an exercise, you should rewrite `SynchAgentBehaviour` using a `SequentialBehaviour` where the two steps are component behaviours.

The other agents in the simulation use the "new day" messages to synchronise

their activities as follows. They begin their activities when they receive a "new day" message, and then send a "done" message to the Ticker Agent when they have completed them. They then await a "new day" message before beginning them again. **Note again the use of a `CyclicBehaviour` to listen for "new day" messages**. Example code for a simple simulation agent that does this is shown below.

```
1  import jade.core.AID;
2  import jade.core.Agent;
3  import jade.core.behaviours.Behaviour;
4  import jade.core.behaviours.CyclicBehaviour;
5  import jade.core.behaviours.WakerBehaviour;
6  import jade.domain.DFService;
7  import jade.domain.FIPAException;
8  import jade.domain.FIPAAgentManagement.DFAgentDescription;
9  import jade.domain.FIPAAgentManagement.ServiceDescription;
10 import jade.lang.acl.ACLMessage;
11 import jade.lang.acl.MessageTemplate;
12
13 public class SimpleSimulationAgent extends Agent {
14   private int day = 0;
15   private AID tickerAgent;
16
17   @Override
18   protected void setup() {
19     //add this agent to the yellow pages
20     DFAgentDescription dfd = new DFAgentDescription();
21     dfd.setName(getAID());
22     ServiceDescription sd = new ServiceDescription();
23     sd.setType("simulation-agent");
24     sd.setName(getLocalName() + "-simulation-agent");
25     dfd.addServices(sd);
26     try{
27       DFService.register(this, dfd);
28     }
29     catch(FIPAException e){
30       e.printStackTrace();
31     }
32     addBehaviour(new DailyBehaviour());
33   }
34
35   @Override
36   protected void takeDown() {
37     //Deregister from the yellow pages
38     try{
39       DFService.deregister(this);
40     }
41     catch(FIPAException e){
42       e.printStackTrace();
43     }
44   }
45
46   public class DailyBehaviour extends CyclicBehaviour {
47
48     @Override
49     public void action() {
50       //wait for new day message
51       MessageTemplate mt = MessageTemplate.MatchContent("new day");
52       ACLMessage msg = myAgent.receive(mt);
53       if(msg != null) {
```

```
54          if(tickerAgent == null) {
55            tickerAgent = msg.getSender(); //AID of the Ticker Agent
56          }
57          //do computation here
58          day++;
59          System.out.println(getLocalName() + "day: " + day);
60          addBehaviour(new WakerBehaviour(myAgent,5000){
61            protected void onWake() {
62              //send a "done" message
63              ACLMessage dayDone = new ACLMessage(ACLMessage.INFORM);
64              dayDone.addReceiver(tickerAgent);
65              dayDone.setContent("done");
66              myAgent.send(dayDone);
67            }
68          });
69        }
70        else{
71          block(); //suspend this behaviour until we receive a
                message
72        }
73      }
74
75    }
76
77  }
```

Listing 2.4: A simple simulation agent that synchronises its daily activities with the Ticker Agent.

2.3 Example: A simulation of buyer and seller agents

To illustrate the use of both agent communication and synchronisation of agent activities through a Ticker Agent, we will consider a simulation in which a buyer agent is given a list of books to buy, and 30 days in which to buy them in. The goal of the buyer agent is to buy all of the books on its list at the cheapest possible price. However, it pays a penalty for each book on its list that it has not bought at the end of the 30 days. The buyer agent can request prices and availability of each book from a number of seller agents on each day. To start in this example, sellers are given a new random collection of books to sell on each day, with random prices. Any books not purchased by the buyer agent by the end of the day are assumed to be returned to the warehouse (or sold to other agents outside of the system). In a more realistic scenario, sellers would adjust the prices of the books they have based upon demand for them.

The code below provides an initial implementation of the simulation where the buyer agent queries each seller each day for the prices of the books that it is looking for. It then records the lowest price that it has ever seen along with the seller that was offering that price. Your task is to extend the simulation so that on each day the buyer must use a strategy to decide whether to buy any books that are in stock or not, bearing in mind that they will not be available at the same price the next day (they may be available for a higher price, a lower price, or not at all).

We can take a look first at the Ticker Agent for this simulation, which is responsible for synchronising the behaviours of buyers and sellers across days, so that each

buyer and seller agent is acting on the same day of the simulation.

```
1  package set10111.simulation;
2
3  import java.util.ArrayList;
4
5  import jade.core.AID;
6  import jade.core.Agent;
7  import jade.core.behaviours.Behaviour;
8  import jade.domain.DFService;
9  import jade.domain.FIPAException;
10 import jade.domain.FIPAAgentManagement.DFAgentDescription;
11 import jade.domain.FIPAAgentManagement.ServiceDescription;
12 import jade.lang.acl.ACLMessage;
13 import jade.lang.acl.MessageTemplate;
14
15 public class BuyerSellerTicker extends Agent {
16   public static final int NUM_DAYS = 30;
17   @Override
18   protected void setup() {
19     //add this agent to the yellow pages
20     DFAgentDescription dfd = new DFAgentDescription();
21     dfd.setName(getAID());
22     ServiceDescription sd = new ServiceDescription();
23     sd.setType("ticker-agent");
24     sd.setName(getLocalName() + "-ticker-agent");
25     dfd.addServices(sd);
26     try{
27       DFService.register(this, dfd);
28     }
29     catch(FIPAException e){
30       e.printStackTrace();
31     }
32     //wait for the other agents to start
33     doWait(5000);
34     addBehaviour(new SynchAgentsBehaviour(this));
35   }
36
37   @Override
38   protected void takeDown() {
39     //Deregister from the yellow pages
40     try{
41       DFService.deregister(this);
42     }
43     catch(FIPAException e){
44       e.printStackTrace();
45     }
46
47   }
48
49   public class SynchAgentsBehaviour extends Behaviour {
50
51     private int step = 0;
52     private int numFinReceived = 0; //finished messages from other
         agents
53     private int day = 0;
54     private ArrayList<AID> simulationAgents = new ArrayList<>();
55
56     public SynchAgentsBehaviour(Agent a) {
57       super(a);
58     }
```

```
59
60      @Override
61      public void action() {
62        switch(step) {
63        case 0:
64          //find all agents using directory service
65          DFAgentDescription template1 = new DFAgentDescription();
66          ServiceDescription sd = new ServiceDescription();
67          sd.setType("buyer");
68          template1.addServices(sd);
69          DFAgentDescription template2 = new DFAgentDescription();
70          ServiceDescription sd2 = new ServiceDescription();
71          sd2.setType("seller");
72          template2.addServices(sd2);
73          try{
74            DFAgentDescription[] agentsType1  = DFService.search(
                  myAgent,template1);
75            for(int i=0; i<agentsType1.length; i++){
76              simulationAgents.add(agentsType1[i].getName()); // this
                    is the AID
77            }
78            DFAgentDescription[] agentsType2  = DFService.search(
                  myAgent,template2);
79            for(int i=0; i<agentsType2.length; i++){
80              simulationAgents.add(agentsType2[i].getName()); // this
                    is the AID
81            }
82          }
83          catch(FIPAException e) {
84            e.printStackTrace();
85          }
86          //send new day message to each agent
87          ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
88          tick.setContent("new day");
89          for(AID id : simulationAgents) {
90            tick.addReceiver(id);
91          }
92          myAgent.send(tick);
93          step++;
94          day++;
95          break;
96        case 1:
97          //wait to receive a "done" message from all agents
98          MessageTemplate mt = MessageTemplate.MatchContent("done");
99          ACLMessage msg = myAgent.receive(mt);
100         if(msg != null) {
101           numFinReceived++;
102           if(numFinReceived >= simulationAgents.size()) {
103             step++;
104           }
105         }
106         else {
107           block();
108         }
109
110       }
111     }
112
113     @Override
114     public boolean done() {
```

```
115       return step == 2;
116     }
117
118
119     @Override
120     public void reset() {
121       super.reset();
122       step = 0;
123       simulationAgents.clear();
124       numFinReceived = 0;
125     }
126
127
128     @Override
129     public int onEnd() {
130       System.out.println("End of day " + day);
131       if(day == NUM_DAYS) {
132         //send termination message to each agent
133         ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
134         msg.setContent("terminate");
135         for(AID agent : simulationAgents) {
136           msg.addReceiver(agent);
137         }
138         myAgent.send(msg);
139         myAgent.doDelete();
140       }
141       else {
142         reset();
143         myAgent.addBehaviour(this);
144       }
145
146       return 0;
147     }
148
149
150
151   }
152
153 }
```

Listing 2.5: The Ticker Agent to synchronise the simulation.

Our Ticker Agent has one multi-step behaviour. Step 1 of this simulates a cyclic behaviour by waiting to receive a "done" message from all buyer and seller agents. It finishes when it has received the required number of messages (unlike a pure cyclic behaviour which would permanently remain in the agent's behaviour queue).

The outline code for the ticker agent class is shown above. It first finds all buyer and seller agents by the querying the Directory Facilitator agent. It then sends a "new day" message to each agent (lines 87-92), and advances its own counter of the current day (it needs to know the current day so that it knows when to terminate the simulation). In the next step, it waits to receive a "done" message from each buyer and seller. Note the use of a message template on line 98 to ensure that only "done" messages are matched. Message templates can be passed as a parameter to the `receive` method of agent, so that only messages in the agent's message queue that match that template are processed. Note also the use of calling `block()` on the behaviour to avoid wasting CPU cycles until a message matching the template arrives.

When a message has been received from every buyer and seller agent the `done()`

method of the behaviour returns `true`, which triggers execution of `onEnd()` as the behaviour is removed from the agent's behaviour queue. This does one of two things depending on the state of the simulation. If the end of the simulation has been reached, e.g. 30 days, then it sends a "terminate" message to each agent, and then calls `doDelete()` on the ticker agent itself to end the simulation. Alternatively, if the end state of the simulation has not been reached then the `reset()` method is called and the behaviour then re-adds itself to the ticker agent, starting again at step 0 of finding buyers and sellers and sending a "new day" message to them.

Finally, note the call to the `doWait()` method of `Agent` on line 33. The parameter passed is the number of milliseconds that the agent should be paused for. Here we wait for 5 seconds before starting to give all of the buyer and seller agents time to launch.

The code for the first part of the buyer agent is shown below.

```
1  package set10111.simulation;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5
6  import jade.core.AID;
7  import jade.core.Agent;
8  import jade.core.behaviours.Behaviour;
9  import jade.core.behaviours.CyclicBehaviour;
10 import jade.core.behaviours.OneShotBehaviour;
11 import jade.core.behaviours.SequentialBehaviour;
12 import jade.domain.DFService;
13 import jade.domain.FIPAException;
14 import jade.domain.FIPAAgentManagement.DFAgentDescription;
15 import jade.domain.FIPAAgentManagement.ServiceDescription;
16 import jade.lang.acl.ACLMessage;
17 import jade.lang.acl.MessageTemplate;
18
19 public class BuyerAgent extends Agent {
20   private ArrayList<AID> sellers = new ArrayList<>();
21   private ArrayList<String>  booksToBuy = new ArrayList<>();
22   private HashMap<String,Offer> bestOffers = new HashMap<>();
23   private AID tickerAgent;
24   private int numQueriesSent;
25   @Override
26   protected void setup() {
27     //add this agent to the yellow pages
28     DFAgentDescription dfd = new DFAgentDescription();
29     dfd.setName(getAID());
30     ServiceDescription sd = new ServiceDescription();
31     sd.setType("buyer");
32     sd.setName(getLocalName() + "-buyer-agent");
33     dfd.addServices(sd);
34     try{
35       DFService.register(this, dfd);
36     }
37     catch(FIPAException e){
38       e.printStackTrace();
39     }
40     //add books to buy
41     booksToBuy.add("Java for Dummies");
42     booksToBuy.add("JADE: the Inside Story");
43     booksToBuy.add("Multi-Agent Systems for Everybody");
44
45     addBehaviour(new TickerWaiter(this));
```

```
46    }
47
48
49    @Override
50    protected void takeDown() {
51      //Deregister from the yellow pages
52      try{
53        DFService.deregister(this);
54      }
55      catch(FIPAException e){
56        e.printStackTrace();
57      }
58    }
59
60    public class TickerWaiter extends CyclicBehaviour {
61
62      //behaviour to wait for a new day
63      public TickerWaiter(Agent a) {
64        super(a);
65      }
66
67      @Override
68      public void action() {
69        MessageTemplate mt = MessageTemplate.or(MessageTemplate.
              MatchContent("new day"),
70              MessageTemplate.MatchContent("terminate"));
71        ACLMessage msg = myAgent.receive(mt);
72        if(msg != null) {
73          if(tickerAgent == null) {
74            tickerAgent = msg.getSender();
75          }
76          if(msg.getContent().equals("new day")) {
77            //spawn new sequential behaviour for day's activities
78            SequentialBehaviour dailyActivity = new
                  SequentialBehaviour();
79            //sub-behaviours will execute in the order they are added
80            dailyActivity.addSubBehaviour(new FindSellers(myAgent));
81            dailyActivity.addSubBehaviour(new SendEnquiries(myAgent))
                  ;
82            dailyActivity.addSubBehaviour(new CollectOffers(myAgent))
                  ;
83            dailyActivity.addSubBehaviour(new EndDay(myAgent));
84            myAgent.addBehaviour(dailyActivity);
85          }
86          else {
87            //termination message to end simulation
88            myAgent.doDelete();
89          }
90        }
91        else{
92          block();
93        }
94      }
95
96    }
97
98    public class FindSellers extends OneShotBehaviour {
99
100     public FindSellers(Agent a) {
101       super(a);
```

```
102     }
103
104     @Override
105     public void action() {
106       DFAgentDescription sellerTemplate = new DFAgentDescription();
107       ServiceDescription sd = new ServiceDescription();
108       sd.setType("seller");
109       sellerTemplate.addServices(sd);
110       try{
111         sellers.clear();
112         DFAgentDescription[] agentsType1  = DFService.search(
              myAgent,sellerTemplate);
113         for(int i=0; i<agentsType1.length; i++){
114           sellers.add(agentsType1[i].getName()); // this is the AID
115         }
116       }
117       catch(FIPAException e) {
118         e.printStackTrace();
119       }
120
121     }
122
123   }
124
125   public class SendEnquiries extends OneShotBehaviour {
126
127     public SendEnquiries(Agent a) {
128       super(a);
129     }
130
131     @Override
132     public void action() {
133       //send out a call for proposals for each book,
134       //using the Contract Net Protocol.
135       numQueriesSent = 0;
136       for(String bookTitle : booksToBuy) {
137         ACLMessage enquiry = new ACLMessage(ACLMessage.CFP);
138         enquiry.setContent(bookTitle);
139         enquiry.setConversationId(bookTitle);
140         for(AID seller : sellers) {
141           enquiry.addReceiver(seller);
142           numQueriesSent++;
143         }
144         myAgent.send(enquiry);
145
146       }
147
148     }
149   }
150
151   public class CollectOffers extends Behaviour {
152     private HashMap<String,Integer> repliesReceived = new HashMap
        <>();
153     private int numRepliesReceived = 0;
154     private boolean finished = false;
155
156     public CollectOffers(Agent a) {
157       super(a);
158     }
159
```

```
160
161     @Override
162     public void action() {
163       boolean received = false;
164       for(String bookTitle : booksToBuy) {
165         MessageTemplate mt = MessageTemplate.MatchConversationId(
              bookTitle);
166         ACLMessage msg = myAgent.receive(mt);
167         if(msg != null) {
168           received = true;
169           numRepliesReceived++;
170           if(msg.getPerformative() == ACLMessage.PROPOSE) {
171             //the reply is an offer so see whether to update the
                  best offer
172             //update if no existing offer
173             if(!bestOffers.containsKey(bookTitle)) {
174               bestOffers.put(bookTitle,
175                   new Offer(msg.getSender(),Integer.parseInt(msg.
                        getContent())));
176             }
177             else {
178               //update only if new offer is better than existing
                    offer
179               int newOffer = Integer.parseInt(msg.getContent());
180               int existingOffer = bestOffers.get(bookTitle).
                    getPrice();
181               if(newOffer < existingOffer) {
182                 bestOffers.remove(bookTitle);
183                 bestOffers.put(bookTitle, new Offer(msg.getSender()
                      ,newOffer));
184               }
185             }
186           }
187
188         }
189       }
190       if(!received) {
191         block();
192       }
193     }
194
195
196
197     @Override
198     public boolean done() {
199       return numRepliesReceived == numQueriesSent;
200     }
201
202     @Override
203     public int onEnd() {
204       //print the offers
205       for(String book : booksToBuy) {
206         if(bestOffers.containsKey(book)) {
207           Offer o = bestOffers.get(book);
208           System.out.println(book + "," + o.getSeller() + "," + o.
                getPrice());
209         }
210         else {
211           System.out.println("No offers for " + book);
212         }
```

```
213       }
214       return 0;
215     }
216
217   }
218
219   public class EndDay extends OneShotBehaviour {
220
221     public EndDay(Agent a) {
222       super(a);
223     }
224
225     @Override
226     public void action() {
227       ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
228       msg.addReceiver(tickerAgent);
229       msg.setContent("done");
230       myAgent.send(msg);
231       //send a message to each seller that we have finished
232       ACLMessage sellerDone = new ACLMessage(ACLMessage.INFORM);
233       sellerDone.setContent("done");
234       for(AID seller : sellers) {
235         sellerDone.addReceiver(seller);
236       }
237       myAgent.send(sellerDone);
238     }
239
240   }
241
242 }
```

Listing 2.6: A buyer agent for a marketplace simulation.

The buyer agent launches a `TickerWaiter` behaviour which listens for messages from the ticker agent to synchronise its activities. If a "terminate" message is received the end of the simulation has been reached and the agent deletes itself. On the other hand, when the agent receives a "new day" message it adds a `SequentialBehaviour` to schedule its daily activities. This is composed of four behaviours that are executed in sequence. The first behaviour finds the AIDs of all seller agents. The second behaviour sends a message to each seller asking it for the price of each book that it would like to buy. It keeps track of the number of queries that it sends, so that it knows how many replies to wait for. To enquire about the cost of books (and ultimately buy them) the buyer uses the Contract Net Protocol, which we will cover in more detail in the lectures. This involves sending a Call For Proposals (`CFP`) to each seller agent, asking them to propose a price for the book. Sellers then reply with either a proposal (using the `PROPOSE` performative), or if they do not have the book in stock they send a `REFUSE` message. In the full simulation the buyer would then send an `ACCEPT_PROPOSAL` message to a seller that it wishes to buy from, and a `REJECT_PROPOSAL` message to all of the other sellers for that book.

The `CollectOffers` behaviour is executed next, and waits for a response from each query sent. It stores the best offers that it has ever received for each book in a `HashMap`, mapping from the book title key to the corresponding offer (implemented through the wrapper `Offer` class that stores the AID of the seller and the price). This behaviour keeps executing until it has received a reply for each query sent. Note the use of the conversation ID to match replies to queries. The `done()` condition is that the number of replies received matches the number of queries sent. The final

behaviour is `EndDay`. This first sends a "done" message to each seller agent to notify them that it has finished with them for the day (sellers do not know that they have finished until they receive confirmation of this from the buyer). The behaviour also sends a "done" message to the ticker agent.

The simple `Offer` wrapper class is as follows.

```
1  package set10111.simulation;
2
3  import jade.core.AID;
4
5  public class Offer {
6    private AID seller;
7    private int price;
8
9    public Offer(AID seller, int price) {
10     super();
11     this.seller = seller;
12     this.price = price;
13   }
14
15   public AID getSeller() {
16     return seller;
17   }
18
19   public int getPrice() {
20     return price;
21   }
22 }
```

Listing 2.7: The Offer wrapper class for the marketplace simulation.

Finally, the seller agent is as shown below.

```
1  package set10111.simulation;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5  import java.util.List;
6
7  import jade.core.AID;
8  import jade.core.Agent;
9  import jade.core.behaviours.Behaviour;
10 import jade.core.behaviours.CyclicBehaviour;
11 import jade.core.behaviours.OneShotBehaviour;
12 import jade.domain.DFService;
13 import jade.domain.FIPAException;
14 import jade.domain.FIPAAgentManagement.DFAgentDescription;
15 import jade.domain.FIPAAgentManagement.ServiceDescription;
16 import jade.lang.acl.ACLMessage;
17 import jade.lang.acl.MessageTemplate;
18
19 public class SellerAgent extends Agent {
20   private HashMap<String,Integer> booksForSale = new HashMap<>();
21   private AID tickerAgent;
22   private ArrayList<AID> buyers = new ArrayList<>();
23   @Override
24   protected void setup() {
25     //add this agent to the yellow pages
26     DFAgentDescription dfd = new DFAgentDescription();
27     dfd.setName(getAID());
28     ServiceDescription sd = new ServiceDescription();
```

```
29      sd.setType("seller");
30      sd.setName(getLocalName() + "-seller-agent");
31      dfd.addServices(sd);
32      try{
33        DFService.register(this, dfd);
34      }
35      catch(FIPAException e){
36        e.printStackTrace();
37      }
38
39      addBehaviour(new TickerWaiter(this));
40    }
41
42    public class TickerWaiter extends CyclicBehaviour {
43
44      //behaviour to wait for a new day
45      public TickerWaiter(Agent a) {
46        super(a);
47      }
48
49      @Override
50      public void action() {
51        MessageTemplate mt = MessageTemplate.or(MessageTemplate.
              MatchContent("new day"),
52              MessageTemplate.MatchContent("terminate"));
53        ACLMessage msg = myAgent.receive(mt);
54        if(msg != null) {
55          if(tickerAgent == null) {
56            tickerAgent = msg.getSender();
57          }
58          if(msg.getContent().equals("new day")) {
59            myAgent.addBehaviour(new BookGenerator());
60            myAgent.addBehaviour(new FindBuyers(myAgent));
61            CyclicBehaviour os = new OffersServer(myAgent);
62            myAgent.addBehaviour(os);
63            ArrayList<Behaviour> cyclicBehaviours = new ArrayList<>()
                  ;
64            cyclicBehaviours.add(os);
65            myAgent.addBehaviour(new EndDayListener(myAgent,
                  cyclicBehaviours));
66          }
67          else {
68            //termination message to end simulation
69            myAgent.doDelete();
70          }
71        }
72        else{
73          block();
74        }
75      }
76
77      public class BookGenerator extends OneShotBehaviour {
78
79        @Override
80        public void action() {
81          booksForSale.clear();
82          //select one book for sale per day
83          int rand = (int)Math.round((1 + 2 * Math.random()));
84          //price will be between 1 and 50 GBP
85          int price = (int)Math.round((1 + 49 * Math.random()));
```

```
86          switch(rand) {
87            case 1 :
88              booksForSale.put("Java for Dummies", price);
89              break;
90            case 2 :
91              booksForSale.put("JADE: the Inside Story", price);
92              break;
93            case 3 :
94              booksForSale.put("Multi-Agent Systems for Everybody",
                    price);
95              break;
96          }
97        }
98
99      }
100
101     public class FindBuyers extends OneShotBehaviour {
102
103       public FindBuyers(Agent a) {
104         super(a);
105       }
106
107       @Override
108       public void action() {
109         DFAgentDescription buyerTemplate = new DFAgentDescription()
                ;
110         ServiceDescription sd = new ServiceDescription();
111         sd.setType("buyer");
112         buyerTemplate.addServices(sd);
113         try{
114           buyers.clear();
115           DFAgentDescription[] agentsType1  = DFService.search(
                myAgent,buyerTemplate);
116           for(int i=0; i<agentsType1.length; i++){
117             buyers.add(agentsType1[i].getName()); // this is the
                  AID
118           }
119         }
120         catch(FIPAException e) {
121           e.printStackTrace();
122         }
123
124       }
125
126     }
127
128   }
129
130   public class OffersServer extends CyclicBehaviour {
131
132     public OffersServer(Agent a) {
133       super(a);
134     }
135
136     @Override
137     public void action() {
138       MessageTemplate mt = MessageTemplate.MatchPerformative(
              ACLMessage.CFP);
139       ACLMessage msg = myAgent.receive(mt);
140       if(msg != null) {
```

```
141         ACLMessage reply = msg.createReply();
142         String book = msg.getContent();
143         if(booksForSale.containsKey(book)) {
144           //we can send an offer
145           reply.setPerformative(ACLMessage.PROPOSE);
146           reply.setContent(String.valueOf(booksForSale.get(book)));
147         }
148         else {
149           reply.setPerformative(ACLMessage.REFUSE);
150         }
151         myAgent.send(reply);
152       }
153       else {
154         block();
155       }
156
157     }
158
159   }
160
161   public class EndDayListener extends CyclicBehaviour {
162     private int buyersFinished = 0;
163     private List<Behaviour> toRemove;
164
165     public EndDayListener(Agent a, List<Behaviour> toRemove) {
166       super(a);
167       this.toRemove = toRemove;
168     }
169
170     @Override
171     public void action() {
172       MessageTemplate mt = MessageTemplate.MatchContent("done");
173       ACLMessage msg = myAgent.receive(mt);
174       if(msg != null) {
175         buyersFinished++;
176       }
177       else {
178         block();
179       }
180       if(buyersFinished == buyers.size()) {
181         //we are finished
182         ACLMessage tick = new ACLMessage(ACLMessage.INFORM);
183         tick.setContent("done");
184         tick.addReceiver(tickerAgent);
185         myAgent.send(tick);
186         //remove behaviours
187         for(Behaviour b : toRemove) {
188           myAgent.removeBehaviour(b);
189         }
190         myAgent.removeBehaviour(this);
191       }
192     }
193
194   }
195 }
```

Listing 2.8: The seller agent for the marketplace simulation.

The seller agent also uses a `TickerWaiter` behaviour to synchronise its daily activities. When it receives a "new day" message it adds four behaviours for that day. The first behaviour, `BookGenerator` adds stock to the agent for that day.

The second behaviour finds the AIDs of all buyer agents. The third behaviour, `OffersServer` does most of the hard work, and listens for calls for proposals from buyers and replies with either an `PROPOSE` or a `REFUSE` message. This is a cyclic behaviour. We add cyclic behaviours to a list, so that we can pass references to them to the `EndDayListener` behaviour. This is because at the end of day, any cyclic behaviours will need to be removed from the agent's behaviour queue so that they can be re-added fresh the following day. This is done by calling `removeBehaviour()` on them in `EndDayListener`. The `EndDayListener` waits for each buyer to send it a "done" message (and is itself a cyclic behaviour). When it has received "done" from each buyer it removes all of the cyclic behaviours (including itself). These will be re-created when the `TickerWaiter` receives a "new day" message from the ticker agent.

Unit 3

Ontologies

In the last chapter we looked at how to send and receive messages between agents. We looked at the FIPA ACL language, which provides the envelope format for our messages. Two important fields on this envelope are the performative and the content. We stressed the importance of choosing the correct performative, as this sets what the purpose of the message is. The performatives available in JADE are specified in the FIPA ACL standard for inter-agent communication.

The performatives that you will use most frequently are:

- REQUEST – ask the other agent to perform an *action* for us
- INFORM – inform the other agent that a fact about the world – *a predicate* – is true
- QUERY_IF – ask the other agent if a fact about the world – *a predicate* – is true
- CONFIRM – reply to a QUERY_IF to confirm that the fact / predicate is true
- DISCONFIRM – reply to a QUERY_IF to confirm that the fact / preducate is not true

There are also additional performatives such as CFP, PROPOSE, REFUSE, ACCEPT_PROPOSAL and REJECT_PROPOSAL, which are used as part of protocols such as the Contract Net. Ultimately, however, these are just macros (shorthand) for various types of REQUEST and INFORM messages. In fact, in the FIPA Agent Communication Language all communication can be done with REQUEST and INFORM performatives, but use of the other performatives can make your code easier to understand.

The performative sets the type of the message – is the message a request or a statement of fact. However, so far we've not specified the meaning of the terms that go inside the content field of the message. Instead, we've been leaving this implicit. For example, we've been sending a message with the title of a book that we would like to buy in the content field of the message (using a CFP or REQUEST performative), and then assuming that the receiving agent knows that the message contents are indeed a book title and not something else like the author or the publisher. This can work when both of the agents are written by us, but in general we would like to be able to write multi-agent systems that are *open*. An open system is one where agents are owned by different individuals, and we don't necessarily know at design time all of the agents (or even the number) that will be in the system. In these cases, we need explicit definitions of words and sentences that agents can use to

communicate. This is what an ontology gives us. At the most fundamental level, an ontology is a shared definition of concepts that your agents will talk about.

3.1 The base ontology in JADE

JADE provides a base ontology which your multi-agent system builds upon. This contains three key classes:

1. concept
2. predicate
3. agent action.

Concepts represent things that we want to talk about. Examples are people, books, CDs, video games, AIDs, CPUs, motherboards, etc. Concepts are composed of *slots*, or *properties*. For example, a person would have properties for name, address, age etc., while a book would have properties for title, author, year of publication etc. **However, it is never legal to send a concept directly in the content field of a FIPA ACL message**. For example it doesn't make sense to ask if a book is true using a QUERY_IF message, or to ask the agent to perform a CPU using a REQUEST message. Concepts only make sense if they appear inside predicates or agent actions. For example, I REQUEST that you sell me this book, where *sell book* is an action that you are REQUESTing the other agent to perform. Or, I INFORM you that the price of this book is 10, where *price (book, 10)* is a predicate (recall that a predicate is a boolean valued function, which our INFORM is asserting is true, i.e. it is a true fact about the world that the book costs 10). Fundamentally, actions change the state of the world, while predicates state that a fact about the world is true.

Predicates, then, are expressions that say something about the world, and which can be either true or false. For example: `(Owns (Book : title "The Hobbit") (Owner : AID "seller 1"))`. This could be sent as a QUERY_IF message to ask if the agent with AID "seller " owns the book with title "The Hobbit". This would evaluate to true if the agent does in fact own the book, and false otherwise. If the evaluation is true, the reply would be sent using the CONFIRM performative, otherwise the reply would be sent using the DISCONFIRM performative. The concepts in the message are Book (with property title), and Owner (with property AID).

Agent actions are actions that an agent can perform, e.g. sell a book, clean the floor, or perform a computation. Typically you want to ask another agent to perform an action for you. To do this you can use the REQUEST performative, or CFP if you are using the Contract Net protocol.

More formally, predicates, concepts and agent actions are the components of valid sentences in the FIPA SL language. The SL language is supported by JADE, and we will use it throughout this module. The grammar of the SL language says that concepts can only appear inside predicates and agent actions. It also specifies that the concept name must appear first inside the concept term, e.g. Book, followed by a colon separated list of properties and their values, e.g. : title "The Hobbit". JADE's content manager handles the creation and parsing of SL language messages automatically once you have an ontology that defines the predicates, concepts and agent actions. All you have to is set the codec (language) in the `setup()` method of your agent.

3.2 Creating an ontology: E-commerce example

To create an ontology, we need to create a class for each concept, predicate and agent action. These should all be placed together in a single package that just contains these classes and nothing else. In the rest of this chapter we will work though development of an ontology for an e-commerce multi-agent system, where agents buy and sell items related to music such as CDs and books about artists. In the problem specification, a seller agent sells CDs and books. Each item that can be bought and sold has a serial number (e.g. on the barcode of the item). In addition to a serial number, CDs have a name and details of each track on them.

- Each track has a title and duration.
- A CD contains at least one track.
- A single contains exactly two tracks.

Books, by contrast, have a title and year of publication in addition to a serial number. Buyers should be able to QUERY sellers to determine whether they have an item with a particular serial number in stock. If the reply is affirmative, the buyer should then be able to place an order, by REQUESTing that the seller sells the item to them.

3.2.1 Step 1: Identify actions, concepts, and predicates

The first step in creating an ontology is to analyse the scenario to identify the actions, concepts, and predicates. We will start with the concepts, which represent things in the world that our agents need to be able to talk about.

Concepts

We can also identify concepts from the scenario. A concept is something that we want our agents to be able to talk about. Concepts have properties. You should think carefully about what should be a concept versus what should be a property of a concept (this is close to the distinction between classes and their properties in object-oriented programming, but with object-oriented programming classes are identified based on the responsibilities they have to deliver part of the functionality of the overall software system, whereas concepts in an ontology correspond to types of thing that agents need to be able to communicate about). In our scenario the concepts would be:

- CD (with properties for serial number, name, and a list of the tracks on the CD),
- Single (with properties for serial number, name, and a list of exactly two tracks),
- Track (with properties for title and duration),
- Book (with properties for serial number, title, and year of publication).

Note that title is, for example, a property of a Book and not a concept in its own right, because it doesn't make sense to talk about a title except in the concept of the book that it is referring to. A concept such as a Book will have multiple *instances*,

corresponding to different books with different property values. It must be stressed though that although this sounds similar to classes and objects in object-oriented programming, conceptually these are very different. The role of classes in object-oriented programming is to structure data together with the methods that change that data (e.g. create a student class with a property for a term-time address and a method to update that term-time address). In an ontology the role of concepts is to identify something that agents need to discuss and define it in a precise way so that every agent has exactly the same understanding of the concept (e.g. so that every agent understands that a book has a serial number, title and year of publication). Although in JADE we must ultimately implement concepts using classes, conceptually they are quite different.

Once we have identified concepts and their properties, we should identify any *constraints* on the properties of each concept. In our scenario these would be:

- CDs must have exactly one name, one serial number, and at least one track.
- Singles must have exactly one name, one serial number, and exactly two tracks.
- Tracks must each have a title and a duration.
- Books must have exactly one title, one serial number, and one year of publication.

This could have the following concepts: Item, Book, CD and track. Here we have identified a hierarchical relation which expresses the fact that CDs and Books have something in common (in this case a serial number), and so are both types (or subclasses) of a more generic Item. The more general concept Item here represents something for sale. Furthermore, a CD is composed of a number of tracks, where each track has properties such as its duration.

Finally, we need to determine any hierarchical (is-a) relations between the concepts we have identified. For example, a Single *is-a* type of CD. Figure 3.1 shows the hierarchical relations between the concepts we have identified. Note that we have additionally identified an Item concept, which captures the commonality between CDs and books, in that both are things that can be bought and sold, and both have a serial number. Because a CD *is-a* concept, this means our agents can put a CD in a message in any place where an Item is needed, likewise for a book.

Finally, note that in the JADE base ontology AID is a concept, which we can use to refer to particular agents in our communications.

Predicates

Recall that predicates represent facts about the state of the world. There is one statement of the fact about the world that our agents need to be able to ascertain in this scenario: whether or not a seller agents *owns* a particular item in stock. Like concepts, predicates also have properties. You can think of a predicate as a boolean function, and its properties as being the parameters of the function. In this case *owns* has two predicates:

- the AID of the agent that owns the item (this is the AID concept shown in Figure **??**), and
- the Item that the agent owns (this is the Item concept shown in Figure **??**).

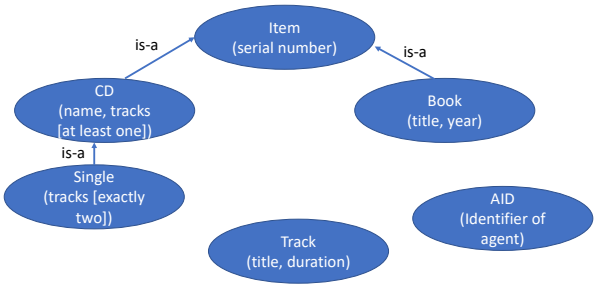

Figure 3.1: The concepts in the e-commerce scenario and the hierarchical relations between them. The properties of a concept are shown in brackets, along with any restrictions on their values.

Like concepts, predicates will have particular instances, e.g. `owns(Agent-X, Metallica)`. Instances of this *owns* predicate will sent from the buyer to the seller in a QUERY_IF message. The seller will then reply with a CONFIRM if it owns that item, and a DISCONFIRM otherwise.

Agent actions

In this scenario there is one action, *sell*. Recall that actions change the state of the world. The only action that changes the state of the world in the above description is a seller selling an item to a buyer. This action would be performed by the seller at the REQUEST of the buyer, i.e. a buyer will send a REQUEST message to the seller, where the content of the message will be the *sell* action. Remember that actions can only go inside a REQUEST (or CFP), and a REQUEST (or CFP) can only taken an action.

The *sell* action will have two properties:

- the AID of the buyer to whom the item is to be sent, and
- the Item that is being sold,

where the Item can be a book or a CD according to the *is-a* relations in Figure 3.1.

3.2.2 Step 2: Code your ontological elements as Java classes

For each concept that you have identified in Step 1, you should create a Java class which implements the `jade.content.Concept` interface, e.g. `Public class Item implements Concept`. The class should have the same name as the concept. For each predicate you should create a class that implements `jade.content.Predicate`, where the class name is the same name as the predicate. For each action you should create a class that implements `jade.content.AgentAction`, where the class name is the same name as the action. These classes should be placed in a common package, e.g. `package set10111.music_shop_ontology.elements;`.

Within each Java class you should have one instance variable for each property that you have identified. This should have the same name as the property and be

a private variable. For properties that are lists, such as the list of Tracks on a CD, the variable should be a `List`.

You must have one getter method and one setter method for each property. If the property name is `X`, then the getter method must be named `getX`, and the setter method must be named `setX`. Note that autogenerate these for you if you write the private instance variables and then right click and choose "Source", "Generate getters and setters...". Note that you should not add a constructor to the class. The class should just use the default constructor that is auto generated by the compiler.

Given the above requirements, the code for the Item class, representing the Item concept, would be as follows:

```
1  package set10111.music_shop_ontology.elements;
2
3  import jade.content.Concept;
4
5  public class Item implements Concept {
6    private int serialNumber;
7
8    public int getSerialNumber() {
9      return serialNumber;
10   }
11
12   public void setSerialNumber(int serialNumber) {
13     this.serialNumber = serialNumber;
14   }
15 }
```

Listing 3.1: The class for the Item concept in our ontology

CDs in our ontology are a type of Item. The "type of" ontological relation is implemented in JADE by Java's class inheritance mechanism. The Java class to represent the concept CD therefore looks like this:

```
1  package set10111.music_shop_ontology.elements;
2
3  import java.util.List;
4  import jade.content.onto.annotations.AggregateSlot;
5  import jade.content.onto.annotations.Slot;
6
7  public class CD extends Item {
8    private String name;
9    private List<Track> tracks;
10
11   @Slot(mandatory = true)
12   public String getName() {
13     return name;
14   }
15
16   public void setName(String name) {
17     this.name = name;
18   }
19
20   @AggregateSlot(cardMin = 1)
21   public List<Track> getTracks() {
22     return tracks;
23   }
24
25   public void setTracks(List<Track> tracks) {
26     this.tracks = tracks;
27   }
```

```
28 }
```

Listing 3.2: The class for the CD concept in our ontology

On line 7 we declare that this extends Item (is a subclass of Item). We then have properties and getters and setters as before. Note that properties can be aggregates, i.e. a CD can have more than one track. We implement aggregate properties as Lists in Java. Note also the use of the `@Slot` annotations discussed above. These are used to specify constraints on the property values that we wish to enforce through our ontology. The annotation `mandatory=true` means that the property value must be entered whenever that concept is inserted into a message. In this case, it means that every time a CD is mentioned its name property must be set before the message is legal. The default is (mandatory=false), which means that specifying the value of that property is optional when creating a message. If you do not specify a mandatory slot then you will get an ontology exception at runtime. The other annotations are `cardMin` and `cardMax`. These are used for aggregate properties (lists), and specify the minimum and maximum number of things that can go in the list. Here we specify that where we talk about the tracks on a CD, there must be at least one track. If this constraint is violated then an ontology exception will also be thrown at runtime. Note that the track property is not mandatory, so we can send messages about CDs without listing all of their tracks.

Which properties are mandatory and which are not is a design decision that depends on the system requirements. We should only make properties mandatory if they must be referenced every time we wish to talk about an instance of that concept (e.g. a particular CD). Otherwise, we should leave the properties as optional to allow more flexibility in how agents talk about the concepts. A complete list of annotations can be found in the Ontology class in the JADE api (in your docs folder). Finally, note that nothing else goes inside our ontological classes except for the properties and their getter and setter methods.

Given the above two examples, the Track class should hopefully look straightforward, and is shown below.

```
1  package set10111.music_shop_ontology.elements;
2  import jade.content.Concept;
3  import jade.content.onto.annotations.Slot;
4
5  public class Track implements Concept {
6    private String name;
7    private int duration;
8
9    @Slot(mandatory = true)
10   public String getName() {
11     return name;
12   }
13   public void setName(String name) {
14     this.name = name;
15   }
16
17   @Slot(mandatory = true)
18   public int getDuration() {
19     return duration;
20   }
21   public void setDuration(int duration) {
22     this.duration = duration;
23   }
24 }
```

Listing 3.3: The class for the Track concept in our ontology

Note that while lists, including ArrayLists, can be included in ontologies, HashMaps cannot. You cannot send a HashMap in the contents of a message.

We also need to create classes for each predicate and agent action (which implement the `Predicate` and `AgentAction` interfaces respectively.

The Owns predicate looks like this, with its two properties and their getters and setters.

```
1  package set10111.music_shop_ontology.elements;
2  import jade.content.Predicate;
3
4  public class Owns implements Predicate {
5    private AID owner;
6    private Item item;
7
8    public AID getOwner() {
9      return owner;
10   }
11   public void setOwner(AID owner) {
12     this.owner = owner;
13   }
14
15   public Item getItem() {
16     return item;
17   }
18   public void setItem(Item item) {
19     this.item = item;
20   }
21 }
```

Listing 3.4: The class for the Owns predicate in our ontology

The Sell action, which we can use to REQUEST an agent to sell us a particular item, looks like this:

```
1  package set10111.music_shop_ontology.elements;
2
3  import jade.content.AgentAction;
4  import jade.core.AID;
5
6  public class Sell implements AgentAction {
7    private AID buyer;
8    private Item item;
9
10   public AID getBuyer() {
11     return buyer;
12   }
13
14   public void setBuyer(AID buyer) {
15     this.buyer = buyer;
16   }
17
18   public Item getItem() {
19     return item;
20   }
21
22   public void setItem(Item item) {
23     this.item = item;
```

```
24    }
25 }
```

Listing 3.5: The class for the Sell agent action in our ontology

Again, we just write code for the properties that the agent action has, in this case the AID of the buyer (i.e. our AID) and the Item that we wish the seller to sell us. Note again that all of our ontological elements should be placed in the same package.

Finally, we need to create an Ontology class to tie all of the elements in the ontology together. This should extend `jade.content.onto.BeanOntology`. This is boiler plate code that you can reuse in all of your projects:

```
1  package set10111.music_shop_ontology;
2
3  import jade.content.onto.BeanOntology;
4  import jade.content.onto.BeanOntologyException;
5  import jade.content.onto.Ontology;
6
7  public class ECommerceOntology extends BeanOntology{
8
9    private static Ontology theInstance = new ECommerceOntology("
       my_ontology");
10
11   public static Ontology getInstance(){
12     return theInstance;
13   }
14   //singleton pattern
15   private ECommerceOntology(String name) {
16     super(name);
17     try {
18       add("set10111.music_shop_ontology.elements");
19     } catch (BeanOntologyException e) {
20       e.printStackTrace();
21     }
22   }
23 }
```

Listing 3.6: The overall ontology class

For those of you that know design patterns, this follows the singleton pattern so that there is only ever one instance of the ontology in the system. This ensures that all agents are using the same ontology, and also saves memory. The crucial line is line 18, which adds all of the ontological elements in the specified package to the ontology.

3.3 Using the ontology

To use our ontology, we have to give all of our agents two new instance variables. The first points to the SL language codec that we will use (supplied by JADE), which gives the grammar of our messages. The second points to the ontology object. We set the value of these variables in the `setup()` method:

```
1  private Codec codec = new SLCodec();
2  private Ontology ontology = ECommerceOntology.getInstance();
3
4  protected void setup() {
5    getContentManager().registerLanguage(codec);
```

```
6     getContentManager().registerOntology(ontology);
7  }
```

Listing 3.7: Setting up the SL codec and ontology

Then, inside of using the `setContent()` method of the ACLMessage class, we instead fill the content field by using `getContentManager().fillContent(msg, content)`. And instead of using `msg.getContent()` to read the message content in the receiving agent, we now use `getContentManager().extractContent(msg)`.

This is illustrated in the behaviour below, which a buyer uses to place an order, i.e. REQUEST that the seller perform the sell action.

```
1  private class PlaceOrder extends OneShotBehaviour{
2
3      protected void action() {
4        // Prepare the action request message
5        ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
6        msg.addReceiver(sellerAID); // sellerAID is the AID of the
             Seller agent
7        msg.setLanguage(codec.getName());
8        msg.setOntology(ontology.getName());
9        // Prepare the content.
10       CD cd = new CD();
11       cd.setName("Synchronicity");
12       cd.setSerialNumber(123);
13       ArrayList<Track> tracks = new ArrayList<Track>();
14       Track t = new Track();
15       t.setName("Every breath you take");
16       t.setDuration(230);
17       tracks.add(t);
18       t = new Track();
19       t.setName("King of pain");
20       t.setDuration(500);
21       tracks.add(t);
22       cd.setTracks(tracks);
23       Sell order = new Sell();
24       order.setBuyer(myAgent.getAID());
25       order.setItem(cd);
26
27       //IMPORTANT: According to FIPA, we need to create a wrapper
             Action object
28       //with the action and the AID of the agent
29       //we are requesting to perform the action
30       //you will get an exception if you try to send the sell
             action directly
31       //not inside the wrapper!!!
32       Action request = new Action();
33       request.setAction(order);
34       request.setActor(sellerAID); // the agent that you request to
              perform the action
35       try {
36        // Let JADE convert from Java objects to string
37        getContentManager().fillContent(msg, request); //send the
             wrapper object
38        send(msg);
39       }
40       catch (CodecException ce) {
41        ce.printStackTrace();
42       }
43       catch (OntologyException oe) {
44        oe.printStackTrace();
```

```
45        }
46
47      }
48
49
50    }
```

Listing 3.8: Use of the ontology to place an order (request that the seller performs the sell action)

Line 37 illustrates the use of `getContentManager().fillContent()` to set the message contents when we are using an ontology. On line 23 we create the agent action from the ontology (on the proceeding lines we create the CD concept that we wish to order). Note carefully lines 32-34. We have to put the AgentAction from our ontology (Sell) inside an Action wrapper object, which contains the AgentAction and the AID of the agent that we are requesting performs the action. You always have to do this when you send an action, as the FIPA specifications require that a message with an action in contains not just the action but the AID of the agent that should perform it.

The code for the seller agent to receive and process the message is shown below. Note the use of a cyclic behaviour so that the seller can receive orders at any time, and process as many as arrive.

```
1  private class SellBehaviour extends CyclicBehaviour{
2      @Override
3      public void action() {
4        //This behaviour should only respond to REQUEST messages
5        MessageTemplate mt = MessageTemplate.MatchPerformative(
             ACLMessage.REQUEST);
6        ACLMessage msg = receive(mt);
7        if(msg != null){
8          try {
9            ContentElement ce = null;
10           System.out.println(msg.getContent()); //print out the
                 message content in SL
11
12           // Let JADE convert from String to Java objects
13           // Output will be a ContentElement
14           ce = getContentManager().extractContent(msg);
15           if(ce instanceof Action) {
16             Concept action = ((Action)ce).getAction();
17             if (action instanceof Sell) {
18               Sell order = (Sell)action;
19               Item it = order.getItem();
20               // Extract the CD name and print it to demonstrate
                     use of the ontology
21               if(it instanceof CD){
22                 CD cd = (CD)it;
23                 //check if seller has it in stock
24                 if(itemsForSale.containsKey(cd.getSerialNumber()))
                       {
25                   System.out.println("Selling CD " + cd.getName());
26                 }
27                 else {
28                   System.out.println("You tried to order something
                         out of stock!!!! Check first!");
29                 }
30
31               }
```

```
32              }
33
34            }
35          }
36
37          catch (CodecException ce) {
38            ce.printStackTrace();
39          }
40          catch (OntologyException oe) {
41            oe.printStackTrace();
42          }
43
44        }
45        else{
46          block();
47        }
48      }
49
50    }
```

Listing 3.9: Use of the ontology to receive a sell action (request that the seller performs the sell action). Note the use of a cyclic behaviour to receive messages at any time.

Note on line 5 the use of a message template, which specifies that this behaviour will only process REQUEST messages. In general you would typically have one behaviour for each type of message that you expect to receive. You should look at the `MessageTemplate` class in the JADE API to see what else you can filter messages by (e.g. by sender or conversation ID). Filtering messages is fundamental to programming in JADE. Lines 14-19 show how the content of the message is extracted and interpreted through the ontology, in contrast to the ad-hoc string passing that we have been using previously.

3.3.1 Sending the result of an agent action

The result of an agent action must be delivered as a predicate. One approach is to have an explicit predicate for this in your ontology. For example, you could have a predicate called *isDelivered(book)*, confirming that the state of the world has changed as a result of the book being delivered.

An alternative approach is to use the Result class in the `jade.content.onto.basic` package. A Result object has two properties. The first is the `AgentAction` object that represents that action whose result is being sent. This can be set by calling `setAction()`. The second property is the result of this action, which can be any `Object`, and is set using `setValue()`. The receiver of the message can call `getAction()` and `getValue()`, respectively, after casting the message content to a Result object.

3.3.2 JADE tools for debugging

JADE contains two very useful tools for debugging. The first is the Sniffer. This shows you graphically the sequence of messages being sent and received between agents in real time while your system is running. To start this, go to the main JADE window and click on Tools, then Start Sniffer. You should then click the arrow next to AgentPlatforms, ThisPlatform, and then Main-Container. This will

expand to show all the agents that are alive. You should right click on agents that you are interested in and then click "Do sniff this agent". A sequence diagram will then be drawn in real time as the agents that you are sniffing send messages to each other. An example is shown in Figure 3.2.

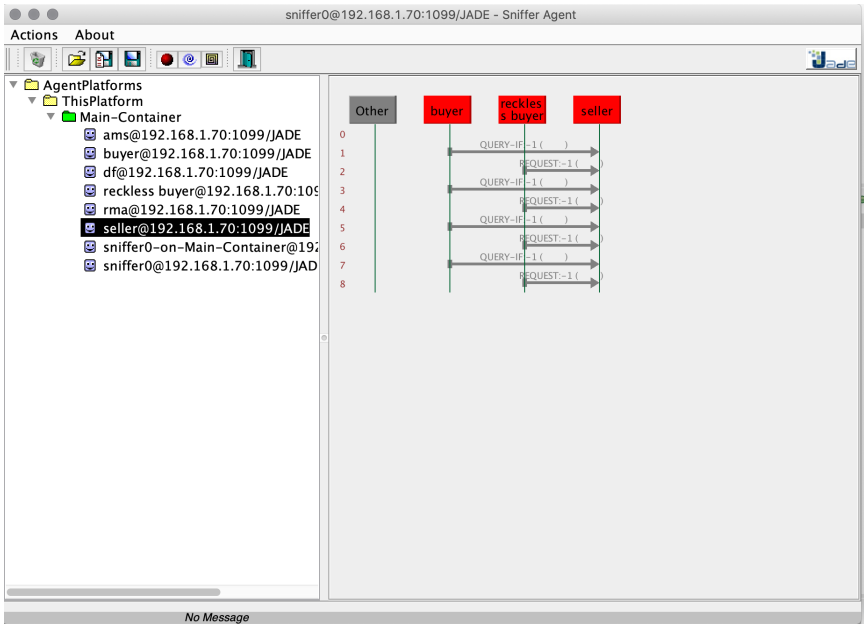

Figure 3.2: Use of the JADE sniffer for observing agent communications.

To see the contents of a message, you should right click on the line in the dequence diagram and choose view message. An example is shown in Figure 3.3.

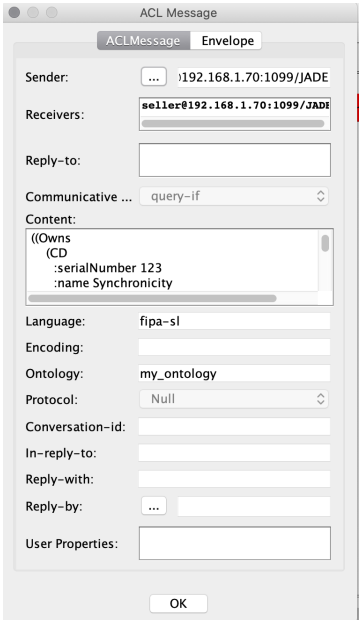

Figure 3.3: Viewing a message in the JADE sniffer.

If your JADE program crashes it can be useful to run the program again and watch it with the sniffer. The last message sent before the program freezes will show you at which state the system is in. To allow time to start the sniffer you may need to put a call to `doWait(20000)` in the setup method of your agent(s). This would pause the system when it is starting and allow you time to sniff all of the relevant

agents. If you have a ticker agent to synchronise behaviour then the call to `doWait` should go in its setup method.

Another useful tool for debugging is the introspector agent. This can be started by clicking tools, start introspector agent. You should again expand AgentPlatforms, ThisPlatform, and MainContainer. You should then right click on the agent that you wish to debug and choose debug on. You will then see the agent's behaviour queue, with the behaviour that is currently executing highlighted. You can click on debug and then step to step through the agent's behaviours one by one. You can also view all of the messages in the agent's message queue, including the complete past history. If your program freezes then by seeing which behaviour was the last to execute and the message queue history you should be more easily able to narrow down what went wrong. An example is shown in Figure 3.4.

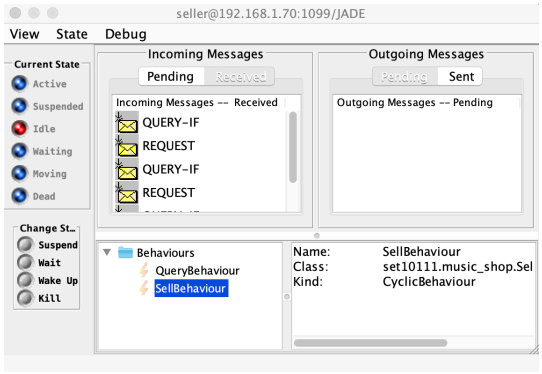

Figure 3.4: The JADE introspector: A powerful tool for debugging.