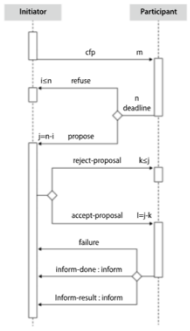

FIGURE 5.19 FIPA Contract Net Protocol. Source: http://www.fipa.org/specs/fipa00029/SC00029H.html

Setting the Stage

One now recovers the scenario initially discussed for the intelligent car and the intelligent factory where one will also consider a few simplifications with the purpose of facilitating the comprehension of the different programming and execution steps. One shall start from a kind of horizontal integration scenario, which shall be described in greater detail now and then complemented with conceptual and technical suggestions on how to extend it to a vertical integration scenario which is left as an exercise.

Horizontal Integration Scenario

The simplified scenario is as follows. Intelligent cars can predict when one of their components is about to fail. When they make such a prediction they will negotiate with several intelligent dealers a time for the repair operation, then they will negotiate with several intelligent factories a date for the production and delivery of the failing component and will place an order for it. Upon producing the part, the factories will ship it to a specific dealer.

Be aware that one is now leaving out some of the most important conceptual complications that were discussed before, addressing them would result in a overwhelmingly complicated example and would transform a didactic example on a leading edge research implementation. The point now is to understand how a basic infrastructure, that could later support such complications, could look like cyber-physically. Doing so is already complicated enough. For now the following additional requirements and constraints apply:

- all the cars are the same;
- each car has four components whose failure it can predict;
- cars predict a failure based on a random number generator that is executed periodically, the same is to say that every time interval

the car randomly selects one of these four components and then randomly determines if it is about to fail or not;

- if a failure is about to occur the car will also randomly predict the when the failure will occur, this prediction will be given in days in the form of an integer from 10 to 30;
- when a dealer in engaged in a negotiation with a car for fixing a repair date it will offer the first available repair slot which is randomized taking into to account the latest possible repair date predicted by the failure predictor;
- a dealer may be able to handle an unlimited number of simultaneous repairs;
- the repair time will vary randomly;
- all the factories are able to produce all the four components;
- certain factories do not produce components but entire cars, these cars will be dynamically added to the environment as they are produced;
- when a factory is engaged in a negotiation process with a car it will offer the first available production slot which is randomized taking into to account the previous negotiations with the dealers.
- factories are able to produce components simultaneously, their capacity is unlimited;
- there can be, at any time, an arbitrary and changeable number of cars, dealers and factories;

Our example focuses mainly on the development of the cyber-component of the different elements considered but the place-holders with the interface between the cyber and the physical will become obvious in the code.

REFERENCE ARCHITECTURE FOR THE HORIZONTAL INTEGRATION SCENARIO

One shall now translate the main ideas of the integration scenario into a more formal engineering framework. In order to do so one shall use the Universal Modelling Language (UML). UML is a general purpose modelling language that has its origins in software engineering. However, UML can be used directly, or can be extended, to accommodate the modelling requirements of virtually any system. A detailed description of the language is off-topic for this book. However, all the main relevant points to the understanding the example will be explained.

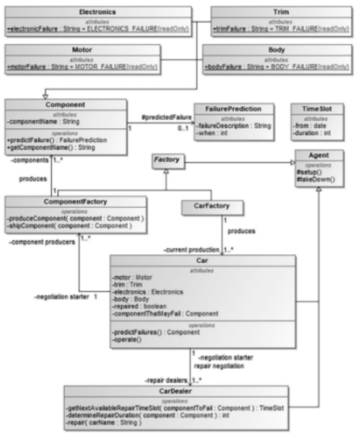

FIGURE 6.1 Structural view of the case.

In order to model the considered scenario there are many important views. One shall now focus on two, the structural view of the system and its behavioural view. The first reflects what the system is, what it looks like, the latter how it should behave. The structural view is presented in Figure 6.1. The diagram in this figure is a UML Class diagram that portraits the main classes of entities in the system and their structural relations.

One can start by analysing the diagram from the top and focusing on the *Component* class. Classes are represented by rectangular shapes with three divisions. The top division contains the name of the class, *Component* in this case , then come two optional compartments. The first

contains the attributes of the class. Attributes are the relevant characteristics that all the objects that belong to the class should have. The second contains the operations of the class. Operations are functions that, when called upon by objects of the class, or even objects from other classes, modify the value of the object's attributes. An object is an instance of a class, that is, an object is an actual thing, it has specific values for its attributes. Operations can generally only be called on objects. Classes are like blueprints for creating many similar objects. There are two ways of representing attributes:

- by including them in the class shape (see for example that the class Car, in the middle of the diagram, has the attributes motor, trim, electronics, body, etc.);
- or by creating a so-called association, represented by the black lines with or without an open arrow in the tip (this is the case of the class Component and the class Failure Prediction where the diagram specifies that one component can have 0 or 1 predicted failures).

With this brief introduction to attributes and operations one should refocus again on the *Component* class. One component has a name, stored in the *componentName* attribute of type String. One now does a bit of repetition, one component can have zero or one *predictedFailure*. One knows this because of the association line and the numbers next to the *Component* and *Failure Prediction* classes. Furthermore, the *Failure Prediction* is not aware of the class *Component*. The attribute *predictedFailure* belongs to the class *Component Prediction*, the same is to say that an object from the class *Component* knows a *predictedFailure* and that it is a *Failure Prediction*. Components have a *predictedFailure* but a *predictedFailure* does not have or know a component. The former is denoted by the open arrow in the association pointing from *Component* to *Failure Prediction*. All the objects derived from the *Failure Prediction* class have two attributes:

- *failureDescription* is a string of characters that describes textually the possible future failure, this is denoted by the notation *failureDescription* **: *String***;
- *when* is an integer that quantifies, in number of days, the time span between now and the moment when the failure is likely to occur.

All the components have one operation called *predictFailure* which is a function that when invoked upon an object from the class component

predicts and returns a predicted failure if there is one. Components also have the operation *getComponentName* to retrieve their name.

Now, one concentrates on the classes connected to the *Component* class by the lines terminating in the white triangle. If the association lines, as explained before, denote a "has a" relation, the line with the white triangle represents a generalization. This is a "is a" relationship. In this context, objects from the classes *Motor*, *Trim*, *Electronics* and *Body* are all components. This means that they have all the attributes and operations of the class *Component* and the attributes and operation of their own classes.

When one examines these classes closely one can see that they have only one attribute each. These attributes are all of type String, they are constant and cannot be modified (denoted by the underlined and *readOnly* notations). These constant attributes are actually storing the values to be used in the *failureDescription* attribute of a *Failure Prediction*.

Back to the *Factory* class, one can see that factories are agents. *Factory* is also a so-called abstract class (denoted by the class name written in italic). One cannot create objects directly from an abstract class. Abstract classes define common operations but are usually incomplete and the implementation of their operations must be described in the classes extending them. One shall have the *Factory* abstract class here as a place holder in case, later on, one wishes to add operations and attributes pertaining to all possible types of factories.

Component Factory and *Car Factory* are specializations of the *Factory* class. One produces components, the other produces cars. One shall leave the *Car Factory* untouched by now and focus on the *Component Factory* that has two operations that simulate different production operations:

- *produceComponent* takes a component as a parameter and triggers all the required physical operations for producing a component as specified by its parameter;
- *shipComponent* simulates the shipping of a component to a dealer.

These two operations are examples of functions that cross the cyber-physical boundary of the system. In a real scenario they would trigger concrete operations in the physical world.

Now, one focuses on the *Car* class which is a quite central concept in this scenario. Each car has four attributes:

- motor — of class *Motor*, abstracting the motor of a car;
- trim — of class *Trim*, abstracting the trim of a car;

- electronics — of class *Electronics*, abstracting the electronics systems of the car;
- body — of class *Body*, abstracting the body of the car.

Each car also has three operations:

- *predictFailures* — analyses all the components in the car and predicts if they are about to fail, the operation returns a component that is about to fail;
- *operate* — simulates the normal operation of the car, its behaviour, which shall be described later.

Notice that in the process of optimizing the production and repair time slots the car will negotiate both with several components producers and repair dealers as denoted by the corresponding associations. A time slot is described in the class with the same name. Such class contains two attributes: *from*, from class date, that contains the starting date of the time slot and *duration*, of type int, whose value describes the duration of the slot in time units.

The *Car Dealer* class is very similar to the *Factory* class. It has several operations that simulate real-world actions. These include:

- *getNextAvailableRepairTimeSlot* — determines the next possible repair slot for a car with a component that is about to fail;
- *determineRepairDuration* — determines the duration of the repair operation;
- *repair* — simulates the actual repair operation.

Both *Car*, *Car Dealer* and *Factory* will be particularly important from the system dynamics perspective as they are the main autonomous components in the scenario. For this reason they all extend the class *Agent*. Consider that, because the the class *Factory* extends the class *Agent*, all the classes extending *Factory* will also be agents.

One is now done with the architectural structural view of the system and can focus on its behavioural view. In the scenario described, objects from the classes *Car*, *Car Dealer* and *Component Factory* will be particularly important from the system's dynamics perspective.

One may start by examining the overall behaviour of the car. One does so by considering a UML activity diagram that summarizes the operations of any car as an active entity in this system (Figure 6.2). The car continuously predicts failures. If a failure is detected then it first negotiates a time slot with a dealer and then one with a component factory. If the negotiation process fails to find a set of compatible slots

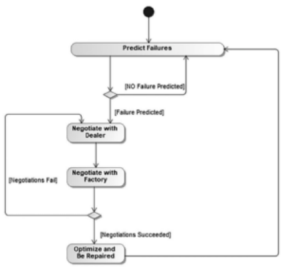

FIGURE 6.2 Main behaviour of the car entity.

then the negotiation restarts until some are found. If compatible slots are found then a selection process takes place and all the participants are notified. This second process is better described as a UML sequence diagram (Figure 6.3). After the car is repaired is resumes the failure prediction cycle.

In order to implement the negotiation and selection processes one shall consider the FIPA Contract Net protocol described earlier. In particular, one will consider a nested version of the protocol where two negotiation processes are intertwined. Figure 6.3 describes this in full detail. Consider that the process occurs between all the dealers and component factories in the system.

Figure 6.3 can be read like this. Upon predicting a failure, the car sends a call for proposals CPF message to all the car dealers. This message contains the information about the component that is predicted to fail. The different dealers will analyse the CFP and determine if they are able to execute the repair. If so, they will send a propose message whose content includes a proposal for the repair time slot. Otherwise, they will send a refuse message, not represented in Figure 6.3, and they leave the negotiation process.

Upon collecting the repair proposals the car will contact all the component factories requiring a production slot for a given component that is

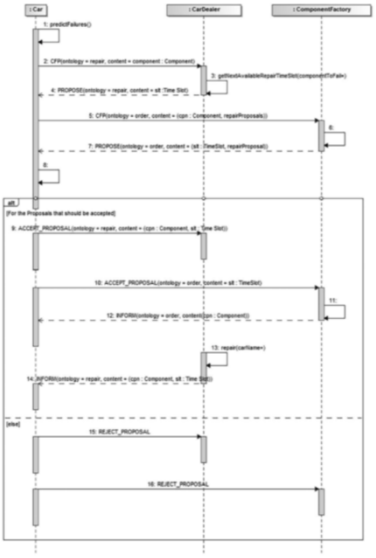

FIGURE 6.3 Message exchange between the different entities.

compatible with the repair slot obtained from the dealers. In order to do this the car sends, as the content of the CFP message to the factories, the information about the failing component and an array with all the possible repair time slots that are under negotiation with the repair dealers.

The factories will then determine if they are able to produce the

component in due time and, if so, send back a proposal with a production time slot. Otherwise they will send a refuse message, not represented in Figure 6.3 and they leave the negotiation process.

When receiving all the production proposals the car determines the set of proposals that leads to a faster repair. It then sends an accept proposal message to the selected dealer and factory simultaneously. It rejects all the other proposals.

For the factory, this signals acceptance of the conditions and the production starts, upon production the car is informed and the part is shipped. Here one assumes that parts are shipped instantaneously and the dealer can start the repair immediately. As soon as the repair is completed the car is notified and resumes its normal operation.

With the main architectural aspects in place one is now better prepared to tackle the programming work. A great deal of the concepts discussed have been modelled in a way that translate very directly into the native agent programming framework considered (JADE). This is precisely what one shall do next, bearing in mind that there may still be loose ends in the architecture, this is almost always the case for any sufficiently large and complex system, but it is also true that a proper architectural design mitigates a great deal of last minute changes and workarounds.

IMPLEMENTATION OF THE HORIZONTAL INTEGRATION SCENARIO

Implementation

One will start now with the activity of writing code and one shall start by the structural aspects. The same is to say that one starts by programming the classes, attributes and operations defined in Figure 6.1. To prepare the incoming programming exercise create an additional package in the solution. This package should be called *book.cpps.mechatronicsagents.horizontal*.

Now comes the implementation of the class *Failure Prediction* which should look like Listing 6.1 and will be used in the implementation of the class *Component*.

Listing 6.1 The Failure Prediction Class.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  public class FailurePrediction {
4
5    private String failureDescription;
6    private int when;
```



```
7
8    public FailurePrediction(String failureDescription, int
       when) {
9      this.failureDescription = failureDescription;
10     this.when = when;
11   }
12
13   public String getFailureDescription() {
14     return failureDescription;
15   }
16
17   public int getWhen() {
18     return when;
19   }
20
21 }
```

As one can see, the implementation is really close to the model. The attributes are declared in lines 5 and 6. Do notice that in JAVA names cannot include spaces and, for that reason, the class is called *FailurePrediction* instead of *Failure Prediction*. The function starting in line 8 is called a constructor and is used to create objects from this class. The functions in lines 13 and 17 provide access to the values of the attributes with the same names.

One should now implement the class *Component* which should look like Listing 6.2.

Listing 6.2 The Component Class.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  public class Component{
4
5    protected FailurePrediction predictedFailure;
6    private final String componentName;
7
8    public Component(String componentName) {
9      this.componentName = componentName;
10   }
11
12   public String getComponentName() {
13     return componentName;
14   }
15
16   public FailurePrediction predictFailure() {
17     return null;
18   }
19 }
```


For now one shall leave this class as it is but later one will return to it to further specify the behaviour of the function *predictFailure*.

The implementation continues with the classes *Motor* (Listing 6.3), *Electronics* (Listing 6.4), *Trim* (Listing 6.5) and *Body* (Listing 6.6).

Listing 6.3 The Motor Class.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  public class Motor extends Component {
4
5    public static final String MOTOR_FAILURE = "
       MOTOR_FAILURE";
6
7    public Motor(String componentName) {
8      super(componentName);
9    }
10 }
```

Listing 6.4 The Electronics Class.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  public class Electronics extends Component {
4
5    public static final String ELECTRONICS_FAILURE = "
       ELEC_FAILURE";
6
7    public Electronics(String componentName) {
8      super(componentName);
9    }
10 }
```

Listing 6.5 The Trim Class.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  public class Trim extends Component {
4
5    public static final String TRIM_FAILURE = "TRIM_FAILURE
       ";
6
7    public Trim(String componentName) {
8      super(componentName);
9    }
10 }
```

Listing 6.6 The Body Class.

```
package book.cpps.mechatronicsagents.horizontal;

public class Body extends Component {

  public static final String BODY_FAILURE = "BODY_FAILURE
    ";

  public Body(String componentName) {
    super(componentName);
  }
}
```

The classes specializing the *Component* class are all very similar and each contains only one constant attribute representing the description of a possible fault.

The implementation of the *Time Slot* class is detailed in (Listing 6.7).

Listing 6.7 The Time Slot Class.

```
package book.cpps.mechatronicsagents.horizontal;

import java.util.Date;

public class TimeSlot {

  private Date from;
  private int duration;

  public TimeSlot(Date from, int duration) {
    this.from = from;
    this.duration = duration;
  }

  public Date getFrom() {
    return from;
  }

  public int getDuration() {
    return duration;
  }

}
```

With these classes in place, it is time to implement the *Factory* and its specialized classes *Component Factory* and *Car Factory* (consider Listings 6.8, 6.9 and 6.10 respectively).

Listing 6.8 The Factory Class.

```
package book.cpps.mechatronicsagents.horizontal;

import jade.core.Agent;

public abstract class Factory extends Agent {

}
```

Listing 6.9 The Component Factory Class.

```
package book.cpps.mechatronicsagents.horizontal;

public class ComponentFactory extends Factory {

  private void produceComponent(Component component) {
    System.out.println(this.getLocalName() + " producing
    component " + component.getComponentName());
  }

  private void shipComponent(Component component) {
    System.out.println(this.getLocalName() + " shipping
    component " + component.getComponentName());
  }

}
```

Listing 6.10 The Car Factory Class.

```
package book.cpps.mechatronicsagents.horizontal;

public class CarFactory extends Factory {

}
```

For now the development environment will probably highlight some errors, here and there, across the different classes. This is fine for now as one is just writing skeleton code and the classes need to be revisited, later on, for implementing the behaviour of the different operations.

One shall now consider the *Car Dealer* class (Listing 6.11) which is rather similar to the *Factory* class and where similar programming patterns apply.

Listing 6.11 The Car Dealer Class.

```
package book.cpps.mechatronicsagents.horizontal;

public class CarDealer extends Agent {
```



```

  private TimeSlot getNextAvailableRepairTimeSlot(
    Component componentToFail) {
    return null;
  }

  private int determineRepairDuration() {
    return 0;
  }

  private void repair(String carName) {
    System.out.println("Repairing a component from " +
    carName);
  }
}
```

Finally comes the *Car* class (Listing 6.12)

Listing 6.12 The Car Class.

```
package book.cpps.mechatronicsagents.horizontal;

public class Car extends Agent{

  private Motor motor; //component id = 0;
  private Trim trim; //component id = 1;
  private Electronics electronics; //component id = 2;
  private Body body; //component id = 3;

  private boolean repaired = true;
  private Component compomentThatMayFail;

  public void setRepaired(boolean repaired) {
    this.repaired = repaired;
  }


  private Component predictFailures() {
    return null;
  }

  private void operate() {
  }

}
```

One now needs to implement a supporting class that is not necessarily part of the architecture but that will play a role in exchanging data

between the agents. This class is called *Pair* and its implementation is given in Listing 6.13. The details of this class are relatively advanced for the scope of this book so the code is presented, as is, without any further explanations. For now it suffices to say that objects of this class are able to store pairs of any two objects.

Listing 6.13 The Pair Class.

```
package book.cpps.mechatronicsagents.horizontal;

class Pair<T0, T1> {

  private T0 first;
  private T1 second;

  public Pair(T0 first, T1 second) {
    this.first = first;
    this.second = second;
  }

  public T0 getFirst() {
    return first;
  }

  public void setFirst(T0 first) {
    this.first = first;
  }

  public T1 getSecond() {
    return second;
  }

  public void setSecond(T1 second) {
    this.second = second;
  }

}
```

At the moment the entire structure of the program is in place but one needs to define its behaviour. This means coming back to the different classes and filling in the functions that were left to implement as well as creating others that are required. It is also at this stage that one will transform some of these concepts into autonomous agents and give them negotiation capacity. In other words, one shall be creating the cyber part of the component. From now on, mainly the part of the code where changes are considered is listed.

One shall start again with the *Component* class and particularly with the *predictFailure* operation. When invoked this function will randomly determine if the component has a failure or not. However, now there is an issue, the class component specializes into four classes *Motor*, *Trim*, *Electronics* and *Body* and all of these have particular failures. This means that when one calls the function *predictFailure*, one actually wants to call the corresponding function of the specialized class. If the component is not specialized then the function in the *Component* class is called by default. One starts with this default function which should look like in Listing 6.14.

Listing 6.14 Component generic failure prediction.

```
1  public class Component{
2
3    protected FailurePrediction predictedFailure;
4    protected Random rnd = new Random();
5
6    public FailurePrediction predictFailure() {
7      boolean inFailure = rnd.nextBoolean();
8      if (inFailure) {
9        predictedFailure = new FailurePrediction("Generic
       Failure", rnd.nextInt(20) + 10);
10       return predictedFailure;
11     } else {
12       return null;
13     }
14   }
15 }
```

Listing 6.14 includes the random generation of the failure (line 7) and the generation of the failure prediction with a generic error message and an occurrence interval between 10 and 30 as specified before (line 9). In the absence of a prediction the function returns null (line 11) as an indication that there is not prediction for any failure at the moment.

Now, one shall consider the known failures. One focuses on the *Motor* class as a example and the remaining classes are left as an exercise. Their implementation is, in all, similar. One may make additional assumptions regarding the prediction of engine failures. In this case one wants to assume that engines may fail 10% of the time (Listing 6.15).

Listing 6.15 Motor failure prediction.

```
1  public class Motor extends Component {
2
```



```
3    public static final String MOTOR_FAILURE = "
       MOTOR_FAILURE";
4
5    @Override
6    public FailurePrediction predictFailure() {
7
8      boolean inFailure = rnd.nextInt(10) < 4;
9      if (inFailure) {
10       predictedFailure = new FailurePrediction(
       MOTOR_FAILURE, rnd.nextInt(20) + 10);
11       return predictedFailure;
12     } else {
13       predictedFailure = null;
14       return null;
15     }
16   }
17 }
```

The 10% failure rate is given partly by the expression in line 8. Do note, however, that one is cheating a bit here as the expression in line 8 gives a failure probability of around 40%, however, the car will, later on, and cheating again, select randomly, and with a uniform distribution, one of the four components. That means a selection probability of 25% thus the probability of being selected and predict a failure is 25% of 40%. The other relevant change in Listing 6.14 is in line 10 where the motor failure specific string from line 1 is used. **Before proceeding one must implement the corresponding functions in classes *Trim*, *Electronics* and *Body*.** Failing to do so will cause them to output the the generic error message. Feel free to modify the failure probabilities.

Now, one comes back to the *Car* class and particularly to the function *predictFailures* which should select randomly one component and its prediction on if it is about to fail. Do note that this is of course a massive simplification of real problem. The JAVA code for this is detailed in Listing 6.16.

Listing 6.16 Car failure prediction.

```
1    private Motor motor; //component id = 0;
2    private Trim trim; //component id = 1;
3    private Electronics electronics; //component id = 2;
4    private Body body; //component id = 3;
5
6    Random rnd = new Random();
7
8    private Component predictFailures() {
```



```
9      FailurePrediction prediction = null;
10     int componentID = rnd.nextInt(4);
11     switch (componentID) {
12       case 0:
13         prediction = motor.predictFailure();
14         if (prediction != null) {
15           return motor;
16         }
17         break;
18       case 1:
19         prediction = trim.predictFailure();
20         if (prediction != null) {
21           return trim;
22         }
23         break;
24       case 2:
25         prediction = electronics.predictFailure();
26         if (prediction != null) {
27           return electronics;
28         }
29         break;
30       case 3:
31         prediction = body.predictFailure();
32         if (prediction != null) {
33           return body;
34         }
35         break;
36       default:
37         return null;
38     }
39     return null;
40   }
```

This means that cars now have a process, even if fictitious and simple in this case, to predict failures within their components. Now it is time to start making cars more autonomous by specifying when to run these processes and when to engage in negotiation. To start this process one needs to make each car an agent. Recall that using the JADE platform this means extending the class agent (see Listing 6.17).

Listing 6.17 Car class extends the Agent class from JADE.

```
1  import jade.core.Agent;
2  import java.util.Random;
3
4  public class Car extends Agent{
5          rest of the code here...
6  }
```


Throughout the code, specially when extending new classes, one may need to add the appropriate import. In this case one has added the import in line 1. The development environment will in most cases automatically add it. However, in other cases, it will highlight parts of the code as an error and ask the user to input an appropriate import. For the sake of space one will not mention all the imports that should be added as the development environment generally provides a very good support for this.

The next step is adding the setup function for the car agent (see Listing 6.18). One can recall what this function is for by reviewing the material in the previous section.

Listing 6.18 Car agent setup function.

```
1  @Override
2  protected void setup() {
3
4  }
```

Now one assumes that a car should use the failure prediction algorithm every 5 seconds. This means creating a periodic behaviour that calls the function (predictFailures) in Listing 6.19 every 5 seconds.

Listing 6.19 Car active failure prediction.

```
1    @Override
2    protected void setup() {
3      System.out.println("The car with name " + this.
       getLocalName() + " is active now!");
4
5      addBehaviour(new TickerBehaviour(this, 5000) {
6        @Override
7        protected void onTick() {
8          Component cpn = predictFailures();
9          if (cpn != null) {
10           System.out.println(cpn.toString() + " from
       class " + getClass().getSimpleName() + " was selected
       ...");
11           System.out.println("The component will have the
        following failure "
12               + cpn.predictedFailure.
       getFailureDescription()
13               + " in " + cpn.predictedFailure.getWhen() +
        " days!");
14         } else {
```

```
15            System.out.println("No failure prediction for
        any component");
16          }
17        }
18      });
19    }
```

In Listing 6.19 one highlights the addition of the *TickerBehaviour* to the agent in line 5. The *TickerBehaviour* is a special behaviour that executes its on *onTick* operation every time period defined in its constructor (see line 5, particularly *new TickerBehaviour(this, 5000)* where the number 5000, defined in milliseconds, is the periodicity of the execution of the behaviour). The body of the *onTick* operation calls the previously defined *predictFailures* function and prints the results on the screen.

One may now execute this agent to see its output. In order to do so a new execution configuration needs to be created similarly to what has been done in the previous section. The details are similar but the arguments configuration should now be *-host localhost car:book.cpps.mechatronicsagents.horizontal.Car* to reflect the fact that now we are starting an agent of type *book.cpps.mechatronicsagents.horizontal.Car* with the name *car*. After executing the new configuration one sees an output similar to the one listed in Listing 6.20.

Listing 6.20 Car active failure prediction output.

```
1  The car with name car is active now!
2  book.cpps.mechatronicsagents.horizontal.Motor@5977a3b3
       from class was selected...
3  The component will have the following failure
       MOTOR_FAILURE in 28 days!
4  No failure prediction for any component
5  No failure prediction for any component
6  No failure prediction for any component
7  book.cpps.mechatronicsagents.horizontal.Trim@48ed0ec6
       from class was selected...
8  The component will have the following failure
       TRIM_FAILURE in 27 days!
9  No failure prediction for any component
10 book.cpps.mechatronicsagents.horizontal.Body@2fb5e48b
       from class was selected...
11 The component will have the following failure
       BODY_FAILURE in 21 days!
12 ...
```


One must recall, however, from Figure 6.2, that there are other things cars must do when detecting a failure. In particular, there is a whole negotiation process to be considered when one failure is detected. Consider that, here again, one simplifies by assuming only one failure at the time. Even with this simplification the development gets sufficient complex has one must suspend further failure detections and start negotiation processes.

Let one start by the negotiation part. To do so one shall use the intertwined FIPA Contract Net protocols earlier detailed in Figure 6.3, where one notes that cars take the role of initiating the protocol. JADE offers a special behaviour for supporting this called *ContractNetInitiator*. In order to use this behaviour one must create a new class called *ProductionRepairNegotiation* that extends *ContractNetInitiator* (see Listing 6.21).

Listing 6.21 Car production and repair negotiation protocol skeleton.

```
1  package book.cpps.mechatronicsagents.horizontal;
2
3  import jade.core.Agent;
4  import jade.lang.acl.ACLMessage;
5  import jade.proto.ContractNetInitiator;
6
7  public class ProductionRepairNegotiation extends
       ContractNetInitiator {
8
9    public ProductionRepairNegotiation(Agent a, ACLMessage
       cfp) {
10     super(a, cfp);
11   }
12
13 }
```

Note in Listing 6.21 that line 9 is the function used to construct objects of this type and takes as an argument the initial message. This initial message is the one triggering the entire protocol in Figure 6.3. After the message is sent the agent is expecting several possible reply messages. Most of these replies will contain time slots referring to production and repair times. Cars will generally want to select the "faster slot" and this is the one that finishes earlier. The same is to say that the value of the attribute *from*, that is the starting date, plus the value of the attribute *duration* is as small as possible. It does not matter if it starts earlier, the car wants it to finish earlier. This means that one must now find a way

of comparing time slots and this implies changes to the *Time Slot* class as detailed in Listing 6.22.

Listing 6.22 Making time slots comparable on their earlier finish.

```
1  public class TimeSlot implements Comparable<TimeSlot> {
2
3    private Date from;
4    private int duration;
5
6    public TimeSlot(Date from, int duration) {
7      this.from = from;
8      this.duration = duration;
9    }
10
11   public Date getFrom() {
12     return from;
13   }
14
15   public int getDuration() {
16     return duration;
17   }
18
19   public static Date addDaysToDate(Date date, int days) {
20     Calendar calendar = Calendar.getInstance();
21     calendar.setTime(date);
22     calendar.add(Calendar.DAY_OF_MONTH, days);
23     return calendar.getTime();
24   }
25
26   @Override
27   public int compareTo(TimeSlot o) {
28     return addDaysToDate(from, duration).compareTo(
       addDaysToDate(o.getFrom(), o.getDuration()));
29   }
30
31 }
```

In Listing 6.22 one must note the following relevant changes. The class now implements the *Comparable<TimeSlot>* interface (line 1) and also the two additional functions. The first, in line 19, adds the duration to the date and computes the finishing date. The second, in line 27, compares two time slots based on their finishing date.

What is more, since the time slots will be sent across the network of agents their information must be serializable. That is, one must be able to covert it to a binary representation suitable for networks and then back to

its original format. This requires implementing the *Serializable* interface and creating two additional functions. Their description is off-topic for this text so they are provided, as is, in Listing 6.23.

Listing 6.23 Making time slots serializable.

```
1  public class TimeSlot implements Comparable<TimeSlot>,
       Serializable {
2
3          ...rest of the code here
4
5    @Override
6    public int hashCode() {
7      int hash = 5;
8      hash = 53 * hash + Objects.hashCode(this.from);
9      hash = 53 * hash + this.duration;
10     return hash;
11   }
12
13   @Override
14   public boolean equals(Object obj) {
15     if (this == obj) {
16       return true;
17     }
18     if (obj == null) {
19       return false;
20     }
21     if (getClass() != obj.getClass()) {
22       return false;
23     }
24     final TimeSlot other = (TimeSlot) obj;
25     if (this.duration != other.duration) {
26       return false;
27     }
28     if (!Objects.equals(this.from, other.from)) {
29       return false;
30     }
31     return true;
32   }
33 }
```

A closer and new inspection of Figure 6.3 shows that in fact more concepts need to be serializable. Messages will include time slots, but also components and often a combination of both. This practically means that one must make the classes *Component* and *Pair* serializable. The code for doing this is provided, as is, in Listings 6.24 and 6.25 respectively.

Listing 6.24 Making components serializable.

```
1  public class Component implements Serializable {
2
3          ...rest of the code here
4
5    @Override
6    public int hashCode() {
7      int hash = 5;
8      hash = 53 * hash + Objects.hashCode(this.
       predictedFailure);
9      return hash;
10   }
11
12   @Override
13   public boolean equals(Object obj) {
14     if (this == obj) {
15       return true;
16     }
17     if (obj == null) {
18       return false;
19     }
20     if (getClass() != obj.getClass()) {
21       return false;
22     }
23     final Component other = (Component) obj;
24     if (!Objects.equals(this.predictedFailure, other.
       predictedFailure)) {
25       return false;
26     }
27     return true;
28   }
29
30 }
```

Listing 6.25 Making pairs serializable.

```
1  class Pair<T0, T1> implements Serializable {
2
3    ...rest of the code here
4
5     @Override
6    public int hashCode() {
7      int hash = 3;
8      hash = 17 * hash + Objects.hashCode(this.first);
9      hash = 17 * hash + Objects.hashCode(this.second);
10     return hash;
11   }
```



```
12
13   @Override
14   public boolean equals(Object obj) {
15     if (this == obj) {
16       return true;
17     }
18     if (obj == null) {
19       return false;
20     }
21     if (getClass() != obj.getClass()) {
22       return false;
23     }
24     final Pair<?, ?> other = (Pair<?, ?>) obj;
25     if (!Objects.equals(this.first, other.first)) {
26       return false;
27     }
28     if (!Objects.equals(this.second, other.second)) {
29       return false;
30     }
31     return true;
32   }
33
34 }
```

Now, one returns to the interaction protocol initiated in Listing 6.21 that is responsible for starting the overall negotiation process. Its final form is detailed in Listing 6.26 (where for the sake of space the packaging and importing details are omitted).

Listing 6.26 Car production and repair negotiation protocol.

```
1  public class ProductionRepairNegotiation extends
       ContractNetInitiator {
2
3    private final Car carAgent;
4
5    public ProductionRepairNegotiation(Agent a, ACLMessage
       cfp, Component componentToFail) {
6      super(a, cfp);
7      this.carAgent = (Car) a;
8      registerHandleAllResponses(new
       RepairProposalsEvaluation(a, componentToFail));
9    }
10
11   @Override
12   protected void handleInform(ACLMessage inform) {
13     System.out.println("Repair has been done");
```



```
14       carAgent.setRepaired(true);
15     }
16   }
```

Noteworthy changes include line 3 where a reference to the carAgent, used later in line 14 to notify the car that the repair has been completed. Line 8 marks the starting of the additional, and intertwined, negotiation protocol for the evaluation of repair proposals and subsequent production. The meaning of line 8 may be a bit overwhelming at the moment. It is using a particular programming pattern of the JADE platform whereby the processing of the repair proposals is being delegated to another behaviour. Finally, in line 14 one has added a handler that is called when the entire negotiation process finishes.

In a simplified way, this behaviour is starting the repair negotiation process and then telling the system that its continuation is handled by the behaviour *RepairProposalsEvaluation* which shall be detailed next. Before going into the programming details one must understand what this new behaviour should be doing. In particular, it shall have different processing steps that need to be fulfilled sequentially. These steps are described in Figure 6.4.

The implementation of the full behaviour is complex and it will be considered step-wisely. One shall start with the attribute declaration and initialization sections that will be provided as is in Listing 6.27. The actual meaning of these attributes and complementary functions will be described when they are used later on.

Listing 6.27 Attributes' declaration and initialization section.

```
1  public class RepairProposalsEvaluation extends
       SimpleBehaviour {
2
3    private static final int EVALUTE_PROPOSALS = 0;
4    private static final int ISSUE_CFP = 1;
5    private static final int WAIT_FOR_NEGOTIATION = 2;
6    private static final int END = 3;
7    private static final int FAIL = 4;
8
9    private int state = EVALUTE_PROPOSALS;
10   private ArrayList<ACLMessage> allProposes = new
       ArrayList<>();
11   private Component componentToFail;
12   private boolean negotiationComplete = false;
13   private Pair<TimeSlot, ACLMessage> selectedProposal =
       null;
14   private boolean done = false;
```


FIGURE 6.4 Proposal Selection and Production Negotiation, the RepairProposalsEvaluation Behaviour.

```
15   private ProductionRepairNegotiation
       productionRepairNegotiation;
16
17   public RepairProposalsEvaluation(Agent a, Component
       componentToFail) {
18     super(a);
19     this.componentToFail = componentToFail;
20   }
21
22   public void setSelectedProposal(Pair<TimeSlot,
       ACLMessage> selectedProposal) {
23     this.selectedProposal = selectedProposal;
24   }
25
26   public void setNegotiationComplete(boolean
       negotiationComplete) {
27     this.negotiationComplete = negotiationComplete;
28   }
29
30 }
```

Now one shall consider the step-based structure detailed before in Figure 6.4 whose implementation is in Listing 6.28.

Listing 6.28 Control of the different execution steps.

```
1  public class RepairProposalsEvaluation extends
       SimpleBehaviour {
2
3          //attribute declaration and initialization code
               goes here...
4
5    @Override
6    public void action() {
7      switch (state) {
8        case EVALUTE_PROPOSALS:
9          //evaluate proposals code goes here...
10         break;
11       case ISSUE_CFP:
12         //issue cfp code goes here...
13         break;
14       case WAIT_FOR_NEGOTIATION:
15         //wait for negotiation code goes here...
16         break;
17       case FAIL:
18         //fail code goes here...
19         break;
20       case END:
21         //end code goes here...
22         break;
23     }
24   }
25
26   @Override
27   public boolean done() {
28     return done;
29   }
30
31 }
```

Listing 6.28 shows a simple behaviour whose *setup* function contains the state machine controlling the different steps discussed before, and the function *done*, which controls the termination of the overall behaviour, is governed by the value of the boolean variable *done*. The different states are labelled using one of the four constants defined in 6.27 namely (EVALUTE_PROPOSALS, ISSUE_CFP, WAIT_FOR_NEGOTIATION, FAIL and END). One shall now consider the code that belongs to the *EVALUATE_PROPOSALS* state which is detailed in Listing 6.29.

Listing 6.29 Collecting and sorting of repair proposals.

```
1  case EVALUTE_PROPOSALS:
2          Vector<ACLMessage> replies = (Vector) this.
       getParent().getDataStore().get(((ContractNetInitiator
       ) this.getParent()).ALL_RESPONSES_KEY);
3          for (ACLMessage msg : replies) {
4            if (msg.getPerformative() == ACLMessage.PROPOSE
       ) {
5              allProposes.add(msg);
6            }
7          }
8
9          allProposes.sort((ACLMessage o1, ACLMessage o2)
       -> {
10           try {
11             return ((TimeSlot) o1.getContentObject()).
       compareTo((TimeSlot) o2.getContentObject());
12           } catch (UnreadableException ex) {
13             Logger.getLogger(RepairProposalsEvaluation.
       class.getName()).log(Level.SEVERE, null, ex);
14           }
15           return 0;
16         });
17
18         System.out.println("Sorted Repair Proposals");
19         allProposes.forEach((msg) -> {
20           try {
21             System.out.println("From " + ((TimeSlot) msg.
       getContentObject()).getFrom() + " with duration " +
       ((TimeSlot) msg.getContentObject()).getDuration());
22           } catch (UnreadableException ex) {
23             Logger.getLogger(RepairProposalsEvaluation.
       class.getName()).log(Level.SEVERE, null, ex);
24           }
25         });
26
27         state = ISSUE_CFP;
28         break;
```

Line 2 in Listing 6.29 is where the replies containing potential proposals for the repair time are retrieved. The ones containing actual proposals are then filtered in lines 3–7 and subsequently sorted in lines 9–16. The additional lines print the sorted proposals on the screen. Finally, after all of this has been done the behaviour goes to the state labelled as *ISSUE_CFP* detailed in Listing 6.30.

Listing 6.30 Triggering factory negotiations.

```
1  case ISSUE_CFP:
2          try {
3            ACLMessage productionCFP = new ACLMessage(
       ACLMessage.CFP);
4            productionCFP.setContentObject(new Pair(
       componentToFail, allProposes));
5            productionCFP.setOntology("Order");
6            DFAgentDescription dfd = new DFAgentDescription
       ();
7            ServiceDescription sd = new ServiceDescription
       ();
8            sd.setName("Factory");
9            sd.setType("Component Factory");
10           dfd.addServices(sd);
11           DFAgentDescription[]
       registeredComponentFactories = DFService.search(
       myAgent, dfd);
12           for (DFAgentDescription dfad :
       registeredComponentFactories) {
13             productionCFP.addReceiver(dfad.getName());
14           }
15           myAgent.addBehaviour(new ProductionNegotiation(
       myAgent, productionCFP, this));
16           state = WAIT_FOR_NEGOTIATION;
17         } catch (IOException | FIPAException ex) {
18           Logger.getLogger(RepairProposalsEvaluation.
       class.getName()).log(Level.SEVERE, null, ex);
19           state = FAIL;
20         }
21         break;
```

There are a few important things to highlight in Listing 6.30. Lines 3–14 create the *CFP* message for negotiating with all the registered factories. The type of message, in this case *CFP*, is set in line 3. Line 4 sets the content of the message. In this case one wants to send a pair containing the specification of the component that is about to fail as well as the list of all possible repair proposals from the previous state. Line 5 specifies the context in which this communication occurs. In this case the car wants to potentially place a production order. Lines 6–14 collect the addresses of all the component factories that are registered in the system and add them as recipients of the *CFP*. Line 15 starts this new and intertwined negotiation process (whose implementation is described later) and line

16 makes this behaviour converge to the next step. Lines 17–20 are JAVA error handling routines which are off-topic at the moment.

The next execution step is all about waiting for the negotiations with the factories and is detailed in Listing 6.31.

Listing 6.31 Waiting for the end of factory negotiations.

```
1  case WAIT_FOR_NEGOTIATION:
2          if (negotiationComplete) {
3            if (selectedProposal == null) {
4              state = FAIL;
5            } else {
6              ACLMessage acceptedRepairProposal =
       selectedProposal.getSecond();
7              Vector<ACLMessage> acceptances = new Vector
       <>();
8
9              for (ACLMessage msg : allProposes) {
10               ACLMessage reply = msg.createReply();
11               try {
12                 if (msg.getContent().equals(
       acceptedRepairProposal.getContent())) {
13                   reply.setContentObject(selectedProposal
       .getFirst());
14                   reply.setPerformative(ACLMessage.
       ACCEPT_PROPOSAL);
15                 } else {
16                   reply.setPerformative(ACLMessage.
       REJECT_PROPOSAL);
17                 }
18               } catch (IOException ex) {
19                 Logger.getLogger(
       RepairProposalsEvaluation.class.getName()).log(Level.
       SEVERE, null, ex);
20               }
21               acceptances.add(reply);
22             }
23
24             this.getParent().getDataStore().put(((
       ContractNetInitiator) this.getParent()).
       ALL_ACCEPTANCES_KEY, acceptances);
25             state = END;
26           }
27         }
28         break;
```

The code in Listing 6.31 executes as follows. When the negotiation process with the factories is completed the value of the attribute *negotiationCompleted* becomes true and the result of the negotiation is that it is not possible to schedule any repair and execution time, then the value of the attribute *selectedProposal* is null. Otherwise, the best proposal, the one that by combining production and repair results in an earlier repair, has already been determined and its value is stored *selectedProposal*. In this case, lines 6–21 ensure that only this proposal is accepted while all the others are rejected. At this stage one is only accepting the repair proposal, the actual acceptance of the production proposal has already taken place in the *ProductionNegotiation* behaviour started in the *ISSUE_CFP* state detailed in Listing 6.30. At this stage also, if everything went as expected, the behaviour converges to the *END* state detailed in Listing 6.32. Here, the done attribute's value is set to true causing the *done* function to terminate the behaviour.

Listing 6.32 Ending the negotiation process.

```
1  case END:
2   done = true;
3   break;
```

Otherwise, the negotiation has failed and in this case the one should implement all the appropriate routines. These should be reflected in the content of the *FAIL* state described in Listing 6.33 whose implementation is left as an additional exercise.

Listing 6.33 Restarting negotiations after failing to find a pair of production and repair time slots.

```
1  case FAIL:
2   //Implement relevant fail routines
3  break;
```

It is now time to concentrate on the *ProductionNegotiation* behaviour started in *ISSUE_CFP* (see Listing 6.30). This behaviour handles effectively the negotiation process with the factories. This behaviour is fully listed in Listing 6.34.

Listing 6.34 Negotiating production.

```
1  public class ProductionNegotiation extends
       ContractNetInitiator {
2
3    private final RepairProposalsEvaluation
       repairProposalsEvaluation;
4    private final ArrayList<ACLMessage> allProposes = new
       ArrayList<>();
```



```
5
6    public ProductionNegotiation(Agent a, ACLMessage cfp,
       RepairProposalsEvaluation repairProposalsEvaluation)
       {
7      super(a, cfp);
8      this.repairProposalsEvaluation =
       repairProposalsEvaluation;
9    }
10
11   @Override
12   protected void handleAllResponses(Vector responses,
       Vector acceptances) {
13     responses.forEach((Object msgObj) -> {
14       ACLMessage msg = (ACLMessage) msgObj;
15       if (msg.getPerformative() == ACLMessage.PROPOSE) {
16         allProposes.add(msg);
17       }
18     });
19
20     allProposes.sort((ACLMessage o1, ACLMessage o2) -> {
21       try {
22         Pair<TimeSlot, ACLMessage> msg1 = ((Pair<TimeSlot
       , ACLMessage>) o1.getContentObject());
23         Pair<TimeSlot, ACLMessage> msg2 = ((Pair<TimeSlot
       , ACLMessage>) o2.getContentObject());
24         TimeSlot ts1 = msg1.getFirst();
25         TimeSlot ts2 = msg2.getFirst();
26         return ts1.compareTo(ts2);
27       } catch (UnreadableException ex) {
28         Logger.getLogger(RepairProposalsEvaluation.class.
       getName()).log(Level.SEVERE, null, ex);
29       }
30       return 0;
31     });
32
33     System.out.println("Sorted Production Proposals");
34     allProposes.forEach((msg) -> {
35       try {
36         Pair<TimeSlot, ACLMessage> msg1 = ((Pair<TimeSlot
       , ACLMessage>) msg.getContentObject());
37         TimeSlot ts1 = msg1.getFirst();
38
39         System.out.println("From " + ts1.getFrom() + "
       with duration " + ts1.getDuration());
40       } catch (UnreadableException ex) {
```



```
41         Logger.getLogger(RepairProposalsEvaluation.class.
       getName()).log(Level.SEVERE, null, ex);
42       }
43     });
44
45     if (!allProposes.isEmpty()) {
46
47       ACLMessage acceptedProduction = allProposes.remove
       (0);
48       try {
49         repairProposalsEvaluation.setSelectedProposal((
       Pair<TimeSlot, ACLMessage>) acceptedProduction.
       getContentObject());
50         ACLMessage replyAccepted = acceptedProduction.
       createReply();
51         replyAccepted.setContentObject(((Pair<TimeSlot,
       ACLMessage>) acceptedProduction.getContentObject()).
       getFirst());
52         replyAccepted.setPerformative(ACLMessage.
       ACCEPT_PROPOSAL);
53         acceptances.add(replyAccepted);
54         System.out.println("Accepted Production offer
       from " + acceptedProduction.getSender().getLocalName
       ());
55       } catch (UnreadableException | IOException ex) {
56         Logger.getLogger(ProductionNegotiation.class.
       getName()).log(Level.SEVERE, null, ex);
57       }
58
59       allProposes.forEach((ACLMessage msg) -> {
60         ACLMessage replyRejected = msg.createReply();
61         replyRejected.setPerformative(ACLMessage.
       REJECT_PROPOSAL);
62         acceptances.add(replyRejected);
63       });
64     } else {
65       repairProposalsEvaluation.setSelectedProposal(null)
       ;
66     }
67     repairProposalsEvaluation.setNegotiationComplete(true
       );
68   }
69
70   @Override
71   protected void handleInform(ACLMessage inform) {
```



```
72     System.out.println(myAgent.getLocalName() + "
       Received inform that the the part has been shipped");
73   }
74
75 }
```

One shall focus on the *handleAllResponses* function in line 12. This function is called when all the factories have made an execution proposal. The variable *responses* contains all the proposals from the factories. Similarly to what has been done before, lines 14–18 filter all responses that actually contain a proposal. Then, also similarly to what has been done before, lines 20–43 sort and print the time slot proposals. Afterwards, the best proposal can be accepted. Do note that it is this behaviour that sets the selected proposal in the *RepairProposalsEvaluationBehaviour* (line 49) and then signals, for the same behaviour, the end of the negotiation process (line 67).

The entire communication protocol from the agent car perspective is now implemented. However, there are still a few bits and pieces missing. The global behaviour of the car needs now to include systematically the detection and repair behaviour, and afterwards the *Car Dealer* and *Factory* classes need to be implemented to be able to engage in these negotiation protocols.

One starts by finishing the *Car* class. In Listing 6.19 one has created a behaviour that every 5 seconds causes the car to predict failures. However, one now needs to go further and guarantee that when a failure is detected, the negotiation and repair processes are triggered. This means that another stepped process is required which shall be implemented as part of the *Car* class *operate* function detailed in 6.35.

Listing 6.35 The operate function.

```
1  private void operate() {
2      switch (state) {
3        case PREDICT:
4          System.out.println("Predicting Failures");
5          compomentThatMayFail = predictFailures();
6          if (compomentThatMayFail != null) {
7            System.out.println(compomentThatMayFail.
       toString() + " from class " + getClass().
       getSimpleName() + " was selected...");
8            System.out.println("The component will have the
        following failure "
9                + compomentThatMayFail.predictedFailure.
       getFailureDescription()
```

```
            + " in " + componentThatMayFail.
predictedFailure.getWhen() + " days!");
        repaired = false;

        System.out.println("Negotiating");
        try {
          ACLMessage cfp = new ACLMessage(ACLMessage.
CFP);
          cfp.setOntology("Repair");
          cfp.setContentObject(componentThatMayFail);
          DFAgentDescription dfd = new
DFAgentDescription();
          ServiceDescription sd = new
ServiceDescription();
          sd.setName("Car Dealer");
          sd.setType("Repair");
          dfd.addServices(sd);
          DFAgentDescription[]
registeredComponentFactories;

          registeredComponentFactories = DFService.
search(this, dfd);
          if (registeredComponentFactories.length != 0)
 {
            for (DFAgentDescription dfad :
registeredComponentFactories) {
              cfp.addReceiver(dfad.getName());
            }
            addBehaviour(new
ProductionRepairNegotiation(this, cfp,
componentThatMayFail));
            state = WAIT_FOR_REPAIR;
          } else {
            System.out.println("Zero registered
factories!");
            state = FAIL;
          }
        } catch (IOException | FIPAException ex) {
          Logger.getLogger(Car.class.getName()).log(
Level.SEVERE, null, ex);
          state = FAIL;
        }
      } else {
        System.out.println("No failure prediction for
any component");
      }
```



```
      break;
    case FAIL:
      System.out.println("Agent Failure");
      doDelete();
      break;
    case WAIT_FOR_REPAIR:
      System.out.println("Waiting for repair");
      if (repaired) {
        state = PREDICT;
      }
      break;
  }
 }
```

The content of the *operate* function includes a state machine with three states: *PREDICT*, *WAIT_FOR_REPAIR* and *FAIL*. The first state is the "normal" state of the system. The car will continuously predict failures and if one prediction emerges then the negotiation process detailed before is started. If an internal error occurs the system will transition to the *FAIL* state and the agent will remove itself from the platform. If the negotiation process is started then the system will wait for its conclusion in the *WAIT_FOR_REPAIR* state. Finally, when the repair is confirmed the system transitions again to the *PREDICT* state. These states are defined as constants and associated to the state variable. A possible implementations for those is in Listing 6.36.

Listing 6.36 State variable and possible values.

```
private static final int PREDICT = 0;
private static final int WAIT_FOR_REPAIR = 1;
private static final int FAIL = 2;

private int state = PREDICT;
```

It is this "new" *operate* function that shall be called within the behaviour cycling every 5 seconds. The final implementation is detailed in Listing 6.37.

Listing 6.37 The car setup function.

```
@Override
 protected void setup() {
   motor = new Motor("Motor of " + this.getLocalName());
   trim = new Trim("Trim of " + this.getLocalName());
   electronics = new Electronics("Electronics of " +
   this.getLocalName());
   body = new Body("Body of " + this.getLocalName());
```



```

   addBehaviour(new TickerBehaviour(this, 5000) {
     @Override
     protected void onTick() {
       operate();
     }
   });

   System.out.println("The car " + this.getLocalName() +
    " is active now!");
 }
```

The implementation of the car is now fully completed. What is left now is to implement the other side of all the communications. One shall start with the car dealer, which is the first actor in the negotiation process (Listing 6.38).

Listing 6.38 The car dealer final implementation.

```
public class CarDealer extends Agent {

 private Random rnd = new Random();

 @Override
 protected void setup() {

   DFAgentDescription dfd = new DFAgentDescription();
   ServiceDescription sd = new ServiceDescription();
   sd.setName("Car Dealer");
   sd.setType("Repair");
   dfd.addServices(sd);
   try {
     DFService.register(this, dfd);
   } catch (FIPAException ex) {
     Logger.getLogger(CarDealer.class.getName()).log(
   Level.SEVERE, null, ex);
   }

   addBehaviour(new ContractNetResponder(this,
   MessageTemplate.and(MessageTemplate.MatchOntology("
   Repair"),
       MessageTemplate.MatchPerformative(ACLMessage.CFP)
   )) {

     @Override
     protected ACLMessage handleCfp(ACLMessage cfp)
```



```
     throws RefuseException, FailureException,
     NotUnderstoodException {
       ACLMessage reply = cfp.createReply();
       try {
         Component componentToFail = (Component) cfp.
     getContentObject();
         reply.setPerformative(ACLMessage.PROPOSE);
         TimeSlot repairTimeSlot =
     getNextAvailableRepairTimeSlot(componentToFail);
         reply.setContentObject(repairTimeSlot);
         System.out.println("Making a repair proposal
     for " + componentToFail.getComponentName() + " from "
      + cfp.getSender().getLocalName());
       } catch (UnreadableException | IOException ex) {
         Logger.getLogger(CarDealer.class.getName()).log
     (Level.SEVERE, null, ex);
         reply.setPerformative(ACLMessage.REFUSE);
       }
       return reply;
     }

     @Override
     protected ACLMessage handleAcceptProposal(
     ACLMessage cfp, ACLMessage propose, ACLMessage accept
     ) throws FailureException {
       ACLMessage reply = accept.createReply();
       reply.setPerformative(ACLMessage.INFORM);
       repair(accept.getSender().getLocalName());
       return reply;
     }
   }
   );

   System.out.println("Car Dealar " + this.getLocalName
   () + " is active now!");
 }

 private TimeSlot getNextAvailableRepairTimeSlot(
   Component componentToFail) {
   Date lastestRepairDate = TimeSlot.addDaysToDate(new
   Date(), componentToFail.predictedFailure.getWhen());
   int repairDuration = determineRepairDuration();
   return new TimeSlot(lastestRepairDate, repairDuration
   );
 }

```

```
57    private int determineRepairDuration() {
58      return rnd.nextInt(4);
59    }
60
61    private void repair(String carName) {
62      System.out.println("Repairing a component from " +
        carName);
63    }
64  }
```

There are two important handlers in Listing 6.38. This first one, in line 23, reacts to all the CFP proposals received and, after analysing them, returns either a concrete proposal for a repair time slot or refuses to continue in the negotiation process. In line 39 the car dealer is reacting to the acceptance of specific proposals. It is worth recalling, a this stage, that when the car accepts a proposal from a dealer, the negotiation with the factory has already occurred. Additionally, and from a conceptual point of view, at this stage in the process the part has already been produced as well. In this simplified scenario, the final confirmation message is immediately sent to the car .

The description of the final version of the component factory follows in Listing 6.39.

Listing 6.39 The component factory final implementation.

```
1  public class ComponentFactory extends Factory {
2
3    private Random rnd = new Random();
4
5    public static Date dateMax(Date d1, Date d2) {
6      if (d1 == null && d2 == null) {
7        return null;
8      }
9      if (d1 == null) {
10       return d2;
11     }
12     if (d2 == null) {
13       return d1;
14     }
15     return (d1.after(d2)) ? d1 : d2;
16   }
17
18   @Override
19   protected void setup() {
20     DFAgentDescription dfd = new DFAgentDescription();
21     ServiceDescription sd = new ServiceDescription();
```



```
22     sd.setName("Factory");
23     sd.setType("Component Factory");
24     dfd.addServices(sd);
25     try {
26       DFService.register(this, dfd);
27     } catch (FIPAException ex) {
28       Logger.getLogger(CarDealer.class.getName()).log(
       Level.SEVERE, null, ex);
29     }
30
31     addBehaviour(
32         new ContractNetResponder(this, MessageTemplate.
       and(MessageTemplate.MatchOntology("Order"),
33             MessageTemplate.MatchPerformative(ACLMessage.
       CFP))) {
34
35       @Override
36       protected ACLMessage handleCfp(ACLMessage cfp)
       throws RefuseException, FailureException,
       NotUnderstoodException {
37         ACLMessage reply = cfp.createReply();
38         try {
39           Pair<Component, ArrayList<ACLMessage>>
           repairProposals = (Pair<Component, ArrayList<
           ACLMessage>>) cfp.getContentObject();
40           int selected = rnd.nextInt(repairProposals.
           getSecond().size());
41           ACLMessage msg = repairProposals.getSecond().
           get(selected);
42           TimeSlot repairSlot = (TimeSlot) msg.
           getContentObject();
43           reply.setPerformative(ACLMessage.PROPOSE);
44           reply.setContentObject(new Pair(new TimeSlot(
           dateMax(TimeSlot.addDaysToDate(repairSlot.getFrom(),
           -rnd.nextInt(20)), new Date()), new Random().nextInt
           (10)), msg));
45         } catch (UnreadableException | IOException ex) {
46           Logger.getLogger(ComponentFactory.class.getName
           ()).log(Level.SEVERE, null, ex);
47           reply.setPerformative(ACLMessage.REFUSE);
48         }
49         return reply;
50       }
51
52       @Override
```



```
53       protected ACLMessage handleAcceptProposal(
         ACLMessage cfp, ACLMessage propose,
54           ACLMessage accept) throws FailureException {
55         ACLMessage reply = accept.createReply();
56         try {
57           reply.setPerformative(ACLMessage.INFORM);
58           Pair<Component, ArrayList<ACLMessage>>
         repairProposals = (Pair<Component, ArrayList<
         ACLMessage>>) cfp.getContentObject();
59           produceComponent(repairProposals.getFirst());
60           shipComponent(repairProposals.getFirst());
61           return reply;
62         } catch (UnreadableException ex) {
63           Logger.getLogger(ComponentFactory.class.getName
         ()).log(Level.SEVERE, null, ex);
64           reply.setPerformative(ACLMessage.FAILURE);
65         }
66         return reply;
67       }
68     }
69     );
70   }
71
72   private void produceComponent(Component component) {
73     System.out.println(this.getLocalName() + " producing
       component " + component.getComponentName());
74   }
75
76   private void shipComponent(Component component) {
77     System.out.println(this.getLocalName() + " shipping
       component " + component.getComponentName());
78   }
79
80 }
```

Aside from the already implemented functions one highlights, in line 5, a function that determines the maximum of two dates. It will be later used to determine the possible scheduling slots. In lines 20–29 the registration of the factory in the service registry of the platform occurs. Its services, become searchable by other agents in the environment, particularly by car agents in the present state. Lines 36–50 include the handling of the call for proposals (CFP) message and the determination of the next available slot. Finally, lines 53–70 include the simulated production start and execution. The implementation of this class concludes the overall

implementation of the exercise and that is left now is to prepare the environment for the final tests.

Final Tests

To test our newly implemented CPPS cyber infrastructure in greater extent one needs to execute it with several cars, dealers and factories. In order to do so one must create or modify existing execution configurations. The instructions for these have been given before when testing a basic agent while setting up of the development environment.

To execute several agents in NetBeans one must add more agents to the configurations. For example adding the following configuration

```
-container
factory1:book.cpps.mechatronicsagents.horizontal.ComponentFactory;
factory2:book.cpps.mechatronicsagents.horizontal.ComponentFactory
```

results in the creation of two component factories named factory1 and factory2 respectively. Do note that the two declarations are separated by a ";" and should be written, in the tool, as single line as opposed to the multiple line declaration above that is solely for illustration purposes. This separation character may vary from operative system to operative system.

One also leaves, as an exercise, for the reader to understand the order by which the different agents should be executed. In an ideal system the order must not matter. In this very simplified case the order still matters and the reader is encouraged to further explore and develop the code to remove this restriction.

As soon as all agents are running the interactions previously developed start to occur. JADE provides a tool called the sniffer agent that allows the visualization of these interactions in runtime. The description of JADE's tools is out of the scope of this book and the reader should carry our their own further investigations to learn more about them. However, for the sake of illustration the trace of the communication in the example are detailed in Figure 6.5.

Figure 6.5 cleary shows the negotiation procedure when several actors of the same kind are included as well as the interactions with the Directory Facilitator to locate the different agents included in the negotiation.

Many things have already been accomplished in the exercise. In particular a simple CPPS has been created to enable different agents to

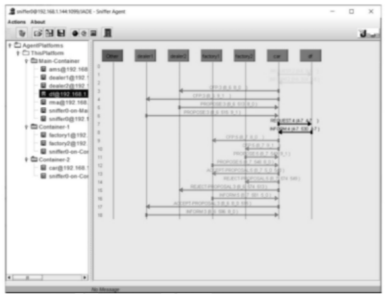

FIGURE 6.5 Trace of the interactions in the example.

negotiate towards the accomplishment of different goals. The exercise, however, has focused very much on a horizontal integration scenario while a great deal of this book discusses design principles that are appropriate for a vertical integration exercise as well. It turns out that such a vertical integration exercise is slightly harder to emulate in a good way. The book should not assume the reader has access to industrial resources to experiment with. Nevertheless, if this is the case, then a new world of further learning opportunities opens up. The following are suggestions on how to modify and/or extend this exercise as well as additional points to consider:

- the interaction patterns explored apply equally regardless of the case;
- a successful CPPS combines horizontal and vertical integration aspects;
- the car factory class has been left to implement and that could be a starting point for an immediate extension.

Taking into account the last point, the car factory shall produce cars and introduce them into the system. In order to do so it must activate in a specific order and with, or without, additional constraints its resources. This resources can be modelled as agents as well. The architectures discussed before provide additional hints on what such modelling could look like. So when one or more orders are received by car factory agents they will need to break them down, optimize them and then engage the resources. Car agents can start their existence already here by driving their own production.

Finally, some additional entity needs to place the orders. It would be easy to image a consumer agent playing this role. The more classes of agents are brought into the scenario the more one must follow proper design procedures. Failing to do so means that the design process discussed along this book is not followed in a connected way. One of the most problematic side-effects of doing so is that a lot of additional system complexity is potentially left without control. This small suggestion on how to extend the exercise will set the reader on the track of the real problems faced by researcher in the field today and will test the limits of the technologies considered in the book. Hopefully, the end of this exercise will be the beginning of another journey for the reader in the exciting world of the FIR.