CHAPTER 5

Preparing the Development Environment

In this chapter one shall prepare the development and runtime environments for the CPPS to be implemented. It is always complicated and risky to provide accurate instructions in a book since the fast-paced technological development of some of the systems required usually dictate that these instructions become obsolete quite quickly. Before proceeding here are some of the things to check and consider:

- this part of the book is designed so that anyone with rightful access to a computer and minimum knowledge on how to operate it should be able to prepare the system, create and run the example. The instructions are given very much in the form of a tutorial that should lead to immediate results if followed carefully. However, if the reader desires to take hers/his understanding one step further, basic notions of object-oriented programming are required, to take advantage of the example beyond just a mere tutorial and to be able to learn from additional sources.
- this guide assumes that one has authorized access to a computer for the purpose of software development and testing and has rightfully acquired administration rights to that computer and its systems;
- it also assumes that the computer is running the windows operating system, the instructions are somehow similar in other operating systems but they are not the same;
- one shall use open source and free software whenever possible. Specific utilization conditions may apply for some of the software considered, particularly if one is using it in a non-personal computer;
- the code described for implementation and testing may directly, or indirectly and unintentionally create security vulnerabilities in the computer hosting it, so you should be in a position where you can assess the security risks resulting from the exercise and you agree to proceed at your own risk and responsibility. The guide will be as transparent as possible but provides absolutely no guarantees in any respect regarding the code.

The previous points are not meant to scare one away from the exercise but mainly to make the point that software development requires a minimum level of computer science knowledge particularly when it comes to security . It is generally a very poor idea to follow instructions, particularly instructions for reconfiguring operating systems or developing and executing code without having a critical thinking understanding of what is going on. With this disclaimer out of the way one now concentrates on the concrete system setup.

Software Setup

There are three different software components that must be set up in order to get our CPPS going:

- the JAVA Virtual Machine — one shall do the programming in the JAVA programming language. JAVA executes in a virtual machine, in a nutshell this means that when you turn your program into instructions that your computer can execute, JAVA turns it into instructions that the virtual machine knows how to execute. The virtual machine will then translate them into the native instruction for your specific computer. As we have discussed before, this means, with limitations, that the same JAVA program may run on any computer that hosts a JAVA virtual machine.
- the Integrated Development Environment — the development environment is a tool that facilitates the development of the programs.
- the base Agent Platform — one shall not develop an agent platform from scratch as they already exist. One shall focus instead the development effort in the cyber-elements that more closely pertain to the CPPS domain.

One shall now setup these components.

SETTING UP THE JAVA VIRTUAL MACHINE

In this example one will use the openJDK virtual machine. The recipe for obtaining it and installing it is as follows:

A go to the openJDK website (https://openjdk.java.net/) and locate the latest windows binaries. They will normally be in the download section.

B download the windows binaries, they are distributed in the form of a zip file.

C verify the integrity of the binaries:

1 the website will provide a SHA checksum string which may look like this *fc7d9eee3c09ea6548b00ca25dbf34a348b3942c815405a1428e0bfef268d08d*.

2 open a Command Prompt in windows, you can do so by clicking the windows icon and writing "*cmd*" followed by the enter key, and navigate to *C:\Windows\System32*.

3 in this directory, execute the following command, considering that the string directoryPath needs to be replaced with the path to the directory where you have saved the downloaded zip file:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All
    rights reserved.
C:\Windows\System32>certutil -hashfile
    directoryPath\openjdk-12.0.1_windows-x64_bin.zip
     sha256
```

4 after executing this command the system will produce a checksum string as a result.

```
SHA256 hash of directoryPath\openjdk-12.0.1_windows
    -x64_bin.zip:
fc7d9eee3c09ea6548b00ca25dbf34a348b3942c815405a1428e

CertUtil: -hashfile command completed successfully.

C:\Windows\System32>
```

If both strings match then the integrity of the downloaded file is verified.

D unzip the file to a specific folder in your system. In the case of the example one will use *C:\Program Files\java\jdk-12.0.1*.

E quick test that the installation is active by navigating to the installation directory and then, within that directory, to the bin folder. Once there, execute the following command:

```
C:\Program Files\java\jdk-12.0.1\bin>java -version
```

The expected output is something similar to:

```
openjdk version "12.0.1" 2019-04-16
OpenJDK Runtime Environment (build 12.0.1+12)
OpenJDK 64-Bit Server VM (build 12.0.1+12, mixed mode,
     sharing)

C:\Program Files\java\jdk-12.0.1\bin>
```

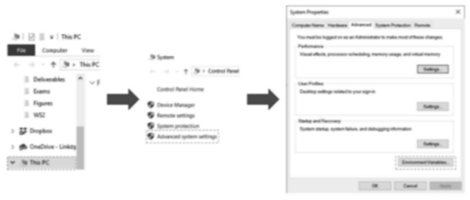
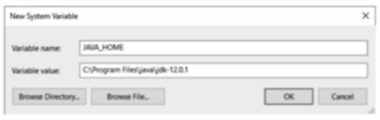


FIGURE 5.1 openJDK setup.

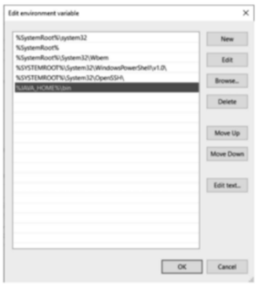

New System Variable

Variable name: JAVA_HOME

Variable value: C:\Program Files\java\jdk-12.0.1

FIGURE 5.2 Defining the *JAVA_HOME* system variable.

F now one needs to make the JAVA installation generally available in the system. One does so by creating additional system environment variables:
1 open the file explorer in windows and locate the *This PC* icon in the left directory tree.
2 right-click it and select *Properties*, the system properties windows pop-up, there select *Advanced system setting* to show the System Properties window, finally click the *Environment Variables...* button (see Figure 5.1 for reference).
3 on the *Environment Variables* window one must create a new system variable called *JAVA_HOME*. The value of that variable should be the directory where one has installed the openJDK which in our case is *C:\Program Files\java\jdk-12.0.1* (see Figure 5.2 for reference).
4 the location of the java related executables needs to be added to


Edit environment variable

```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
%JAVA_HOME%\bin
```

FIGURE 5.3 Editing the ***Path*** system variable.

the *Path* system variable. To do so select it, click edit and then add the *JAVA_HOME* entry precisely as specified. The same is to say %*JAVA_HOME*%*bin* (see Figure 5.3 for reference).
5 click OK on all the opened windows to make the changes effective.
6 now open a new Command Prompt and type:

```
C:\Program Files\java\jdk-12.0.1\bin>java -version
```

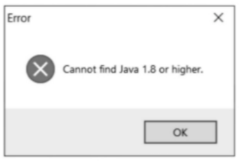
If you see the same output as before then the JAVA Virtual Machine set up process is completed!

Now it is time to install the development environment.

SETTING UP THE DEVELOPMENT ENVIRONMENT

In this example one will use the Apache NetBeans development environment. The recipe for obtaining it and installing it is as follows:

A go to the Apache NetBeans website (https://netbeans.org/) and locate


Error

Cannot find Java 1.8 or higher.

FIGURE 5.4 Common installation error.

the latest windows binaries. They will normally be in the download section.

B download the windows binaries, they are distributed in the form of a zip file.

C verify the integrity of the binaries. This is similar to the process previously considered for the openJDK binaries, although you may have to use a different checksum algorithm, for example SHA-512, check on the website the correct checksum algorithm.

D unzip the file to a specific folder in your system. In the case of the example one will use C:\Program Files\NetBeans\jdk-12.0.1.

E to test that NetBeans is working navigate into the folder where it has been unzipped. This folder should now contain another folder called *netbeans*. Inside that folder navigate to the folder called *bin*. Inside the folder *bin* double click the icon *netbeans.exe* or *netbeans64.exe* depending if you are running a 32 or 64 bit compliant operating system respectively. Most modern computers will be on the second case.

F most likely upon double clicking you will get the error in Figure 5.4. If you do not get this error it most probably means that NetBeans has found another JAVA virtual machine that was already installed in your system. There is nothing wrong with that at this point and some changes may come later.

G if you did get the error then you need to navigate to the *netbeans* folder and then into the *etc* folder. In that folder locate the file *netbeans.conf* and open it for editing. Usually the Notepad application from windows will work.

FIGURE 5.5 Netbeans upon opening.

H with the file open for editing locate the line that contains "*#netbeans_jdkhome = pathToJAVA*". Remove the # and replace the *pathToJAVA* part with the path to the JAVA virtual machine installation. In the case of this example *C:\Program Files\java\jdk-12.0.1*, then save the file.

I if you now click again on the appropriate NetBeans icon in the ...*netbeans\bin* folder, Netbeans should launch properly (see Figure 5.5).

SETTING UP THE AGENT PLATFORM

One shall now set up simultaneously the base JAVA project that uses the agent platform that will serve as a support tool for further developments and the base agent platform itself. This can be done by following these steps:

A start by creating a JAVA application. To do that click in File and then New Project, a window similar to Figure 5.6 shows up. Select JAVA with Maven and then JAVA Application, click next.

B then configure the project similarly to Figure 5.7. It is important to keep the Project Name and the Group ID as indicated in the figure. In the Project Location choose a convenient folder in your system. Leave all the other fields untouched.

C the project is now created and one needs to instruct it to use the base agent platform. To do so one must locate, in the project structure, a file called pom.xml (see Figure 5.8).

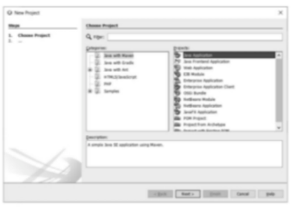


FIGURE 5.6 Project creation.

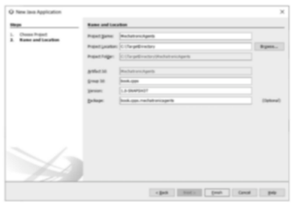


FIGURE 5.7 Project creation.

D now, one must edit this file to instruct the development environment to locate and use the agent platform. One must add one repository and one dependency to the project. The relevant part of the edited file is depicted in Figure 5.9.

The original references to the repository and dependency can be found in https://jade.tilab.com/developers/maven/.

E the final step is to execute a Clean and Build operation in the project. The very first time one executes this operation one must be connected to the Internet. This is not required in any subsequent Clean and Build operations. The process for doing so is depicted in Figure 5.10.

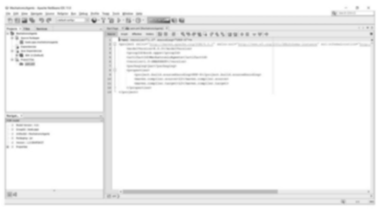
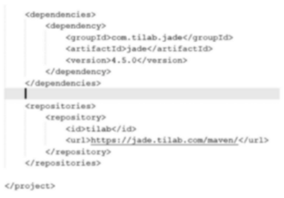


FIGURE 5.8 Opening the POM file.



```
    <dependencies>
        <dependency>
            <groupId>com.tilab.jade</groupId>
            <artifactId>jade</artifactId>
            <version>4.5.0</version>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>tilab</id>
            <url>https://jade.tilab.com/maven/</url>
        </repository>
    </repositories>

</project>
```

FIGURE 5.9 Opening the POM file.

The system is now fully set up to use the JADE agent platform. Before one starts programming there are a few important things to know about JADE mainly regarding the way it is meant to be used, the way it operates and the set of services and tools it provides. One shall only touch these aspects at the surface, for an in-depth training and understanding of them the interested reader is encourage to go on details through JADE's documentation, available in https://jade.tilab.com/ as well as the comprehensive material in [4] and the introduction example in https://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html.

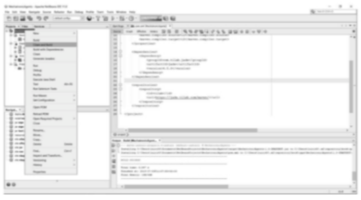


FIGURE 5.10 Clean and Build operation.

JADE IN A NUTSHELL

JADE stands for JAVA Agent Development Framework[1] and therefore offers a programming framework for developing behaviour based agents. Conceptually, the basic model of a JADE agent enables the design of agents belonging to any of the agent approaches discussed before (anywhere from rational to reactive agents). In order to do so JADE provides a runtime environment for agents that mainly includes, among others:

- communication — by offering mechanisms for agents to locate each other and exchange messages with a customizable content;
- mobility — by allowing the agents to migrate between different computational platforms executing JADE;
- introspection — by supporting tools that enable analysing what is happening within agents in the platform in real-time;
- debugging — by supporting tools that enable the analysis of the execution of the agents in the platform.

A JADE agent is therefore as intelligent and autonomous as its behaviours command it to be. It is also technically, from a computational point of view, a thread. A thread is an independent path of execution in a program. A very basic computer program only has one single path of execution. Quite often several actions must occur in parallel, this is accomplished by creating multiple threads. From a JADE point of view this means that all the agents are completely independent from each other from programming and execution perspectives. They are, however,

1 https://jade.tilab.com/

not independent from the base platform that provides them the services discussed. This must be understood in greater detail.

Architecture of the Agent Platform

JADE agents require the presence of a JADE platform in order to operate. This means that the platform must be active in all the computing devices where one intends to execute agents. When the platform is active in a given platform is appears as one agent container. One may think of a container as the digital ecosystem where an agent operates. The whole platform then becomes a collection of containers spread across different computing devices. Containers are responsible for example for transporting the messages between the agents and managing, internally, the agent's life cycle. JADE requires the presence of a special container called the Main Container. The Main Container, in a rather simplistic description, coordinates the activity of other containers. It also provides the following additional services:

- the Directory Facilitator Agent (DF) — which allows other agents to register their services in order for them to be discoverable;
- the Agent Management System Agent (AMS) — which ensures platform consistency by registering and allocating a unique identifier any agent in the platform. The AMS is the only agent in the platform offering services for adding and removing both agents and containers to and from the platform. "Normal" JADE agents can create other agents but may not remove them.

Structurally, an active JADE platform looks like Figure 5.11.

The "Normal" Agents and their Behaviours

One has discussed before that agents are independent executing units. One has also said that agents have behaviours. Before going into further details regarding the different native types of behaviours it is important to understand how agents execute their behaviours. This applies to all behaviours regardless of their type and purpose. Each agent has a fixed behaviour execution path as described in Figure 5.12.

Each agent has a function called *Setup* . This function is only executed once as soon as the agent comes alive in the platform. It is in this function that all the starting behaviours of an agent must be launched. After this function is executed the agent is active and will cyclically operate as

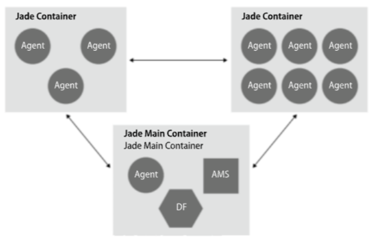


FIGURE 5.11 Structural view of an active JADE platform.

described in Figure 5.12. It will start by assessing if it should remove itself from the platform. "Normal" agents are allowed to remove themselves from the system by calling their *Do Delete* function but they are not allowed to remove other agents (recall from the previous discussion that only the AMS is able to do so). If there is no reason for removing itself then the agent will select one behaviour from its list of active behaviours and will execute it.

All the JADE behaviours have two mandatory functions. The first is called *Action* and should contain all the code representing the main activities that the behaviour should carry out. The second is called *Done* and contains the termination conditions for the behaviour. After selecting one behaviour from the active list, the agent will execute the behaviour's *Action* functions and subsequently its *Done* function. The *Done* function will return a boolean value indicating if the behaviour should be executed again or if it should be removed from the list. The cycle then repeats, continuously until the agent is removed from the platform.

Before removal, the agent will execute the *Take Down* function where all the relevant clean-up operations must be programmed. This is the last opportunity for doing so before the agent leaves the platform.

It is very important to understand this cycle because it has important execution implications from an agent performance perspective. The execution speed of the cycle described in Figure 5.12 directly depends on the execution speed of all the behaviours in the active list of the agent. If this cycle is relatively fast one may almost consider, conceptually, that

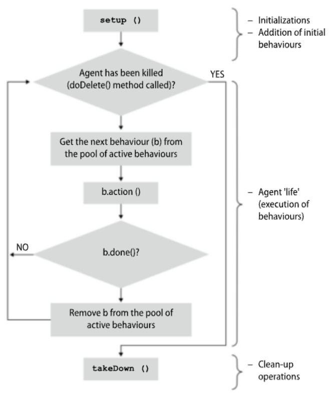

setup ()
– Initializations
– Addition of initial behaviours
Agent has been killed (doDelete() method called)?
YES
Get the next behaviour (b) from the pool of active behaviours
b.action ()
– Agent 'life' (execution of behaviours)
NO
b.done()?
Remove b from the pool of active behaviours
takeDown ()
– Clean-up operations

FIGURE 5.12 JADE Agent Execution Path. Source: [4].

the different behaviours execute in parallel. In practice they execute in round robin order. To maximize the illusion of parallel behaviour execution one must ensure that the code inside the functions *Action* and *Done* is the minimal code required for each execution round of the behaviour. It is also important to consider that, because of this execution cycle, the functions *Action* and *Done* must not contain any blocking code as this will prevent all the other behaviours from executing. Similarly, actions that would individually take a long execution time if executed at once, should be broken down into smaller execution chunks, each of which will be executed per cycle until the main action is completed. Considering these programming guidelines leads to very performing agents. It is also possible to associate behaviour to other threads. This is,

however, an advanced topic in the context of this book and the interested reader is refereed to JADE's documentation for further details.

Before one proceeds to the development of the CPPS scenario it is worth spending some time in some basic agent programming tasks to get familiar with what agent code looks like and be able to understand better the more complex examples coming later.

Programming a very basic Agent

Before one starts it is worth mentioning that there are excellent sources for further learning and these include the comprehensive material in https://www.iro.umontreal.ca/~vaucher/Agents/Jade/JadePrimer.html, https://jade.tilab.com/ and [4]. The previous references are of paramount importance to get a full understanding other than just being able to run the examples in this book.

To start now one must do the following:

- on the JAVA project that was prepared before one should create a new package. One may do this by right-clicking the *Source Packages* icon in the project tree (see Figure 5.13). Name the new package "book.cpps.mechatronicagents.basic".
- one must now create a new JAVA class. The process is similar to the package creation. Now, right-click the new package icon, instead of the *Source Packages* icon, and select new *Java Class*. Name that class *BasicAgent* and leave all the other fields by default.

The newly created class should look like this Listing 5.1.

Listing 5.1 Newly created agent.

```
package book.cpps.mechatronicagents.basic;

public class BasicAgent {

}
```

Now one must modify it as in the Listing 5.2 to create the first very basis agent;

Listing 5.2 Basic agent.

```
import jade.core.Agent;

public class BasicAgent extends Agent {

  @Override
```

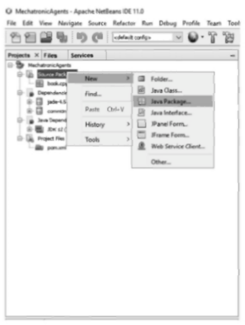


FIGURE 5.13 Creating a new package.

```
  protected void setup() {
    System.out.println("Hello, I am an Agent!");
  }

  @Override
  protected void takeDown() {
    System.out.println("I am out. Bye!");
  }

}
```

This very basic agent does not include any behaviours. Notice that the class extends the base class *Agent* from the package *jade.core.Agent* of the JADE framework. The extended agent prints the string *Hello, I am an Agent!* when started, and the string *I am out. Bye!* when removed from the platform. To see this output one must instruct the development environment to execute this agent. This can be achieved for testing purposes by:

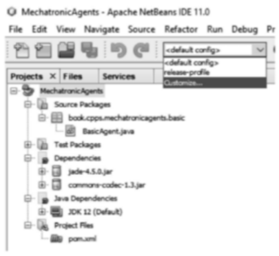


FIGURE 5.14 Creating an execution configuration.

A clicking the configuration drop-down box and selecting *Customize* (see Figure 5.14). After this the window in Figure 5.15 pops up.
B select *Run* on the displayed tree and write *jade.Boot* in *the Main class* field and *-host localhost basicAgent:book.cpps.mechatronic-agents.basic.BasicAgent.*
C click the *OK* button to come back to the development environment, right-click the *MechatronicAgents* icon in the project tree and click run on the expanded menu (see Figure 5.16).

A lot of the program output that is generated at this stage which is connected to the previous actions. When one fills the different fields in Figure 5.15 one is instructing the development environment to start a JADE platform in the computer where it is located. That is the meaning of *jade.Boot* in the *Main class*. After that, one is instructing JADE to create an agent with the name *basicAgent* and from the type *book.cpps.mechatronicagents.basic.BasicAgent*. That is the meaning of *-host localhost basicAgent:book.cpps.mechatronicagents.basic.BasicAgent*.

In response to the *Run* command the development environment does just that (see Listing 5.3). Lines 20–25 and 27–30 show the JADE platform being launched on the system and its main services being activated. Line 26 shows the output of the agent *Setup* function as programmed.

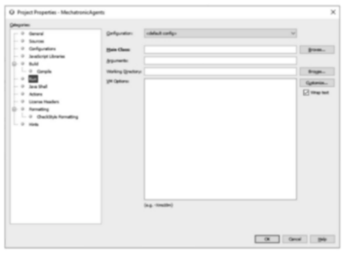


FIGURE 5.15 Creating an execution configuration (continued).

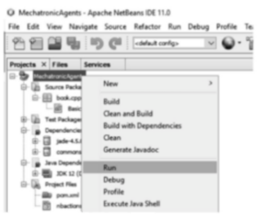


FIGURE 5.16 Executing the basic agent.

Listing 5.3 Generated output.

```
1  juli 04, 2019 1:11:08 EM jade.core.Runtime beginContainer
2  INFO: ----------------------------------
3    This is JADE 4.5.0 - revision 6825 of 23-05-2017
       10:06:04
4    downloaded in Open Source, under LGPL restrictions,
5    at http://jade.tilab.com/
6  ----------------------------------------
7  juli 04, 2019 1:11:09 EM jade.imtp.leap.LEAPIMTPManager
       initialize
8  INFO: Listening for intra-platform commands on address:
9  - jicp://192.168.1.102:1099
10
11 juli 04, 2019 1:11:09 EM jade.core.BaseService init
12 INFO: Service jade.core.management.AgentManagement
       initialized
13 juli 04, 2019 1:11:09 EM jade.core.BaseService init
14 INFO: Service jade.core.messaging.Messaging initialized
15 juli 04, 2019 1:11:09 EM jade.core.BaseService init
16 INFO: Service jade.core.resource.ResourceManagement
       initialized
17 juli 04, 2019 1:11:09 EM jade.core.BaseService init
18 INFO: Service jade.core.mobility.AgentMobility
       initialized
19 juli 04, 2019 1:11:09 EM jade.core.BaseService init
20 INFO: Service jade.core.event.Notification initialized
21 juli 04, 2019 1:11:13 EM jade.mtp.http.HTTPServer <init>
22 INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces
       .internal.jaxp.SAXParserImpl$JAXPSAXParser
23 juli 04, 2019 1:11:13 EM jade.core.messaging.
       MessagingService boot
24 INFO: MTP addresses:
25 http://win00770.ad.liu.se:7778/acc
26 Hello, I am an Agent!
27 juli 04, 2019 1:11:13 EM jade.core.AgentContainerImpl
       joinPlatform
28 INFO: ---------------------------------------
29 Agent container Main-Container@192.168.1.102 is ready.
30 ---------------------------------------
```

In order to terminate the test one must stop all the executing agents and platforms. This is a very important step and without it is is not possible to repeat the test. This can be done by selecting *Run* from the top menu and clicking *Stop Build/Run*. Alternatively, it is also possible to press on the red stop button (see Figure 5.17).

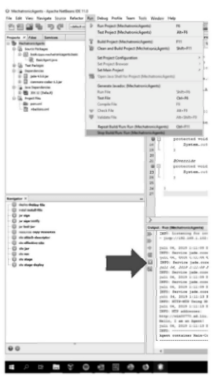


FIGURE 5.17 Stopping a running agent and platform.

Now, one should create a slightly more sophisticated agent. In the new version the agent will display its name, while setting up, and will start to have behaviours. One shall start by defining a behaviour for the agent. In order to do so one must add a new class to the package *book.cpps.mechatronicagents.basic* and name that class *MyOneShotBehaviour*. The contents of that class should be as detailed in Listing 5.4.

Listing 5.4 A behaviour that only executes once.

```
package book.cpps.mechatronicagents.basic;

import jade.core.behaviours.SimpleBehaviour;

```

```java
public class MyOneShotBehaviour extends SimpleBehaviour{

  @Override
  public void action() {
    System.out.println("Executing something!");
  }

  @Override
  public boolean done() {
    System.out.println("This only executes once because
    this function returns true");
    return true;
  }

}
```

There are a few things to highlight regarding Listing 5.4. To create a behaviour one must extend one of the base behaviour classes from JADE. In the example, the class *SimpleBehaviour* is being extended. This extension requires an explicit implementation of the *Action* and *Done* functions discussed before. Because in this implementation of the *Done* function it returns *true* immediately after the first execution cycle, this behaviour will only execute once in accordance to the cycle described in Figure 5.12.

Now, one must instruct the agent to use this behaviour. We will instruct it also to output its name. This entails changes to the *Setup* function of the *BasicAgent* class. These changes are described in Listing 5.5.

Listing 5.5 Adding the behaviour to the agent.

```java
1  @Override
2  protected void setup() {
3    System.out.println("Hello, I am an Agent! My name is " +
       getLocalName());
4    addBehaviour(new MyOneShotBehaviour());
5  }
```

The important point to retain from Listing 5.5 is that in line 3 one uses the function *Get Local Name* to retrieve the name of the agent and in line 4 one uses the function *Add Behaviour* to add a new behaviour to the agent.

If one executes the application now, aside from the JADE initialization output described in Listing 5.3, one also sees new output as detailed in Listing 5.6.

Listing 5.6 Additional output of the agent execution.

```
Hello, I am an Agent!My name is basicAgent
Executing something!
This only executes once because this function returns
    true
```

The result of the *Done* function regulates the termination condition of the behaviour. If one modifies the code in Listing 5.4 by changing the return value to false then the output would be as in Listing 5.7.

Listing 5.7 Additional output of the agent execution when the Done function of the behaviour always returns false.

```
Hello, I am an Agent!My name is basicAgent
Executing something!
This only executes once because this function returns
    true
This only executes once because this function returns
    true
This only executes once because this function returns
    true
This only executes once because this function returns
    true
This only executes once because this function returns
    true
This only executes once because this function returns
    true
This only executes once because this function returns
    true
...
```

This very brief introduction is intended to facilitate the understanding of the incoming CPPS-oriented example where other, additional and more complex, programming patterns and functions will be explored and explained. It is important that one understands these very basic ones before proceeding.

Agent Interaction Protocols

Another important aspect to consider before moving to the implementation work described in the next chapter is that agents normally use well defined interaction protocols. These interaction protocols can be used in different contexts when the agents want to ask other agents to do something, or when they need to negotiate. The Foundation for Intelligent Physical Agents (FIPA) has historically generalized and standardized a

few of these interaction protocols. Two of them are of particular relevance for the technical work ahead.

The first is the FIPA Request Protocol detailed in Figure 5.18. This protocol defines normally an interaction between one agent that initiates the protocol and one or many agents replying to it. It is considered when the initiating agent wishes to ask other agents to execute something. The initiator agent sends a message of type *Request* to the participant agents. The content of that message defines what is being requested. If the participating agents agree to execute the requested action they reply with an *Agree* message and proceed with the execution. The participating agents may also immediately refuse to execute by replying with a *Refuse* message. In this case the interaction terminates immediately.

If an *Agree* as been sent then the initiator agent is expecting either an *Inform* or a *Failure* message as the follow-up of the agreement. The first indicates that the execution succeeded and the last that it has failed.

In all cases the contents of the messages may provide additional information.

The FIPA Contract Net Protocol can be used to support negotiation procedures. It also includes one initiating agent and one or many participants. In this case the initiator sends a *CFP* message, CFP stands for call for proposals. This message invites the participant agents to provide an offer for the action being requested. This will come in the form a reply using a message of type *Propose*. Similarly to the Request Protocol a refusal may be considered at this stage, which will terminate the negotiation.

Upon analysing the received proposals the initiator may accept one or more. This is done by sending an *Accept Proposal* message. All the messages that are not accepted must be rejected. The interaction with the participants that see their proposals being rejected ends here. For the ones whose proposals get accepted then the initiator has the expectation of receiving further messages. This last part of the protocol operates very similarly to the *Request* protocol. It is the same types of messages being exchanged with the same meaning.

Combinations of these two protocols will be extensively used next. Quite complex and useful interaction patterns can be enacted by considering them.

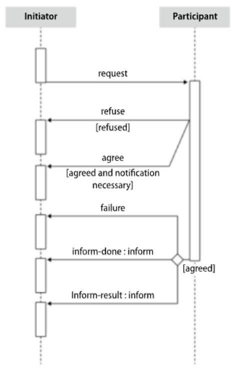


FIGURE 5.18 FIPA Request Protocol. Source: http://www.fipa.org/specs/fipa00026/SC00026H.html

Test Your Understanding

The following questions were designed to cover the main aspects discussed in this chapter. Use them to assess the depth of you understanding so far.

A What is an agent platform and what services does it provide?
B Which processing actions are included in a processing cycle of a JADE agent?
C What should be declared in the setup function of an agent? And in the takeDown function?
D How is a behaviour defined in JADE?
E Which agent interaction protocol would you use to support a negotiation process among JADE agents?