aimpulse intelligent systems

Aimpulse Spectrum Tutorial

Dr. Jan D. Gehrke
Dr. Arne Schuldt

Aimpulse
Intelligent Systems GmbH
Fahrenheitstraße 1
28359 Bremen

Aimpulse Spectrum Tutorial

Contents

1 Introduction 1
2 Hello World 1
3 Behavior-Based Hello World 3
4 Repeated Agent Behavior 4
5 Time and Agent Activation 6
6 Composite Agent Behavior 8
7 Message Exchange 10
8 Interaction Protocols 13
9 Agent Discovery 17

Version: 1.3.2

1 Introduction

Aimpulse Spectrum is a Java-based runtime environment for multiagent systems. This document is intended as a hands-on tutorial for programmers starting to implement their first agents with Spectrum. All examples presented in this document plus a comprehensive implementation specification and the full Application Programming Interface (API) documentation are available on the Internet at developer.aimpulse.com.

Aimpulse Spectrum is shipped in a ZIP file that contains start scripts for both Microsoft Windows and Unix-like systems (in the `bin` directory). With these scripts, Spectrum can be used as a stand-alone solution. As an alternative, it is also possible to use it as a library (in the `lib` directory) in other software systems.

Throughout this tutorial, Aimpulse Spectrum is used as a library. In order to install it in an Integrated Development Environment (IDE), all JAR files contained in the `lib` directory of the ZIP must be added to the build path of the respective project.

The remainder of this tutorial is structured as follows. It starts with a "Hello World" example with one agent in Section 2. Subsequently, Section 3 introduces the behavior concept to further structure agent implementation. Section 4 presents the implementation of repeated agent behavior. Section 5 gives an overview about time and agent activation. The implementation of more complex agent behavior is then dealt with in Section 6.

Attention is then shifted to multiagent systems and agent interaction. Section 7 introduces how messages can be exchanged in Spectrum. Section 8 describes how to structure message exchange based on interaction protocols. The directory as a means to discover other agents is then presented in Section 9.

2 Hello World

This section describes how to implement a minimal agent system with Aimpulse Spectrum. This example system is populated by only one agent. And this agent only writes two words on the console: Hello World.

Listing 2.1 shows the implementation of the `HelloWorldAgent`. It extends the `Agent` class from the `com.aimpulse.spectrum.core` package. Extending this abstract class requires implementing both the `call()` and the `hasTerminated()` methods.

Listing 2.1: Implementation of the `HelloWorldAgent`

```
1  public class HelloWorldAgent extends Agent {
2
3    @Override
4    public Timestamp call() {
5      System.out.println("Hello World");
6      return null;
7    }
8
9    @Override
10   public boolean hasTerminated() {
11     return true;
12   }
13 }
```

The `call()` method in Line 4 contains the actual agent implementation. It is called by the agent platform whenever the agent is executed. Consequently, the command for writing the example text to the console is placed here. With the return value of the `call()` method, an agent can request its next execution time. In this example, one execution suffices. Therefore, the return value is simply `null`.

The `hasTerminated()` method in Line 10 is used by the platform to determine whether the agent has terminated. It is called after each execution of the `call()` method. In this example, the agent has terminated after one execution. Hence, the method returns `true`.

Listing 2.2 shows the wider context of how the new `HelloWorldAgent` is actually executed. In Line 7, the agent platform is instantiated. It is responsible for executing the agents.

Listing 2.2: Implementation of the `HelloWorld` example

```
1  public class HelloWorld {
2
3    private static final Timestamp START = Timestamp.fromMillisecondsSince1970(0);
4    private static final String NAME = "my_agent";
5
6    public static void main(String[] arguments) {
7      Platform platform = new DefaultPlatform();
8      platform.setup(new DiscreteEventSimulation(START));
9      Agent agent = new HelloWorldAgent();
10     platform.addAgent(agent, NAME);
11     platform.start();
12   }
13 }
```

The agent platform has two different execution modes: simulation and operation. In this example, Aimpulse Spectrum is used in its simulation mode. Therefore, the platform is configured for discrete event simulation in Line 8. The constructor of the `DiscreteEventSimulation` class takes one parameter: the start time of the simulation.

Discrete Event Simulation

Discrete event simulation (DES) is a popular and powerful simulation approach. In DES, time advances based on the timestamps of events registered by simulation processes (agents in Aimpulse Spectrum). Thus, instead of normal time (wall-clock time) that advances continuously with constant speed, DES has on-demand step-wise time ticks with dynamic speed. Simulation speed will depend on the distance between events and the time needed to process these events.

Following the platform setup, a `HelloWorldAgent` is instantiated in Line 9. It is added to the platform in Line 10. The second parameter of the `addAgent()` method is the name of the agent. Finally, the platform is started in Line 11. As expected, the output on the console is `Hello World`.

There is another aspect worth mentioning already in this initial example. The start time of the platform is specified in milliseconds since 1970 (the Unix epoch). However, the respective

timestamp is not simply represented by a `long` value. Instead, a specific `Timestamp` value type is used in Line 3.

Value Types

In contrast to primitive types like `int` or `String` a value type is an immutable data type with specific semantics. Value types ensure integrity by value range checks and may provide type-specific operations like adding time as a `Duration` value type to a `Timestamp`. Using value types is a continuous concept throughout Aimpulse Spectrum. The implementation of Spectrum follows the fail-fast principle that reports any failures or illegal states immediately at the interface of the system. Value types support this principle already on the Java compiler level.

3 Behavior-Based Hello World

The example in Section 2 directly uses the interface between the platform and the agent. The advantage of this interface is that agent implementation is not further restricted. A drawback, however, is that dealing with this raw interface is rather challenging. This particularly holds if an agent should be capable of complex behavior. This section extends the previous example to the event-driven agent behavior concept which eases agent implementation significantly.

The `Behavior` interface represents individual parts of agent behavior. A common super class for user-implemented behavior is the abstract `BasicBehavior` class. Listing 3.1 shows the implementation of a `HelloWorldBehavior` that extends the `BasicBehavior`.

Listing 3.1: Implementation of the `HelloWorldBehavior`

```
1  public class HelloWorldBehavior extends BasicBehavior {
2
3    public HelloWorldBehavior(BehaviorController controller) {
4      super(controller);
5    }
6
7    @Override
8    public void action() {
9      System.out.println("Hello␣world");
10   }
11 }
```

In Line 3, the constructor delegates the `BehaviorController` parameter (which is usually the respective agent) to the constructor of the super class. The actual implementation of this behavior can be found in the `action()` method in Line 8. In this example, it simply contains the command for writing the example text to the console. The `action()` method is called by the agent when it executes the behavior.

Listing 3.2 shows the wider context of how the behavior is actually executed. The differences to Listing 2.2 can be found in Lines 4 to 6.

Listing 3.2: Implementation of the behavior-based `HelloWorld` example

```
1   public static void main(String[] arguments) {
2     Platform platform = new DefaultPlatform();
3     platform.setup(new DiscreteEventSimulation(START));
4     Agent agent = new Agent();
5     Behavior behavior = new HelloWorldBehavior(agent);
6     agent.addBehavior(behavior);
7     platform.addAgent(agent, NAME);
8     platform.start();
9   }
```

In Line 4, an `Agent` is instantiated. In contrast to the previous example, it is not necessary to implement a new agent. All implementation has been moved to the respective behavior. Therefore, it suffices to use the `com.aimpulse.spectrum.mas.behaviors.Agent`. Note that, despite of the same name, this `Agent` is from another package than the one used in Section 2.

In Line 5, the `HelloWorldBehavior` from Listing 3.1 is instantiated. The constructor of the behavior takes the agent as a parameter because the agent is its behavior controller. Subsequenlty, the behavior is added to the agent in Line 6. Also this example leads to the expected output on the console: `Hello World`.

4 Repeated Agent Behavior

The example in Section 3 performs only one operation and is then finished. In practice, however, there will often be situations that demand for repeated execution. As an example, consider an agent that writes text to the console over and over again.

Listing 4.1 shows a modified example of the `action()` method from Listing 3.1. It simply places the command for writing the text to the console into an infinite loop. Note, however, that this implementation has some severe drawbacks and is therefore strictly not recommended.

Listing 4.1: Example of a `Behavior` with an inner loop (strictly not recommended)

```
1   @Override
2   public void action() {
3     while (true) {
4       System.out.println("Hello␣world");
5     }
6   }
```

On the one hand, the implementation in Listing 4.1 simply prevents time advancement in discrete event simulation. As soon as the inner loop is reached for the first time, it is never left again. This means that the program flow of the respective thread of execution never returns to the platform (which is responsible for time advancement). Outside of simulation, one might consider this acceptable at first glance because time advancement is implicit in real-world operation.

But on the other hand, this implementation also influences the execution of other behaviors and agents. If an agent with multiple parallel behaviors is stuck in one behavior, the other behaviors

will not be executed again. Furthermore, the platform might be configured to execute multiple agents in one thread (which is a very good idea for reasons of performance). Then, even the execution of the other agents in the same thread can be disrupted. Therefore, programmers are encouraged to prevent (or reduce) time-consuming inner loops wherever possible.

Fortunately, there is a very simple solution to this challenge. Instead of an inner loop, calling `reschedule()` as in Listing 4.2 at the end of the `action()` method has the same desired effect: the `action()` method is executed repeatedly. In between, however, the agent may schedule other behaviors and the platform may schedule other agents. Furthermore, the platform can care for time advancement if this is required in simulation.

Listing 4.2: Example of a `Behavior` with an outer loop (recommended)

```
1   @Override
2   public void action() {
3     System.out.println("Hello␣world");
4     reschedule();
5   }
```

One advantage of the behavior concept is that every behavior can have its own internal state that is kept even over multiple executions. This makes it easy to limit the number of loops to a maximum as in Listing 4.3.

Listing 4.3: Implementation of the `RepeatedBehavior`

```
1  public class RepeatedBehavior extends BasicBehavior {
2
3    private static final int NUM_LOOPS = 3;
4    private int count = 0;
5
6    public RepeatedBehavior(BehaviorController controller) {
7      super(controller);
8    }
9
10   @Override
11   public void action() {
12     System.out.println("Hello␣world");
13     if (++count < NUM_LOOPS) {
14       reschedule();
15     }
16   }
17 }
```

The `reschedule()` method is implemented in the `BasicBehavior` class and can therefore be used in all derived classes. However, it is not always necessary that a behavior is immediately re-scheduled. Instead, it might suffice if the behavior is executed again after some time. The `block(Duration)` method requests re-scheduling after a specified duration. The `blockUntil(Timestamp)` method requests re-scheduling at a specified time.

These auxiliary methods are internally mapped to the respective methods of the `Behavior` interface. The behavior-based agent then, in turn, maps these methods to its interface with the

platform. The Aimpulse Spectrum implementation specification gives a more detailed overview of the event-driven behavior architecture.

Note that it does not matter at which position re-scheduling is requested within the `action()` method. In Listing 4.2, the order of Lines 3 and 4 can be reversed without any influence on the resulting output. Furthermore, a later request for re-scheduling overrides a previous one. In order to cancel a previous re-scheduling request, the `finish()` method can be called.

5 Time and Agent Activation

The previous Section 4 explains how re-scheduling can be requested at a specific `Timestamp` or after a specific `Duration`. Often, the starting point when dealing with time is to retrieve the current time from the system. In plain Java, this is generally done by calling the static method `System.currentTimeMillis()`. However, calling that method is not advisable in simulation. It always returns the current system time. That is usually not the expected result when one wants to retrieve the simulation time in a simulation that runs faster than real-time or has a logical start in, say, the year 1970. This section describes an example of how the time of the (simulated) multiagent system can be accessed instead.

Listing 5.1 shows the implementation of the `TimerBehavior`. In Line 12, the `action()` method requests the `currentTime()` and stores the resulting `Timestamp` in the variable named `now`. The `currentTime()` method is available because the behavior extends the `BasicBehavior`. Other behaviors can access the `currentTime()` method from their respective `BehaviorController`.

Listing 5.1: Implementation of the `TimerBehavior`

```
1  public class TimerBehavior extends BasicBehavior {
2
3    private static final long SECONDS_PER_DAY = 24 * 60 * 60;
4    private static final Duration ONE_DAY = Duration.fromSeconds(SECONDS_PER_DAY);
5
6    public TimerBehavior(BehaviorController controller) {
7      super(controller);
8    }
9
10   @Override
11   public void action() {
12     Timestamp now = currentTime();
13     String formatted = ISO8601_EXTENDED_COMPLETE.format(now);
14     System.out.println("The␣time␣is␣now␣" + formatted);
15     Timestamp nextExecution = now.add(ONE_DAY);
16     blockUntil(nextExecution);
17   }
18 }
```

In Line 13, the `ISO8601_EXTENDED_COMPLETE` format is used to format the `Timestamp` for the console output. It is a static import from the `TimestampFormat` interface. Alternative formats can be defined based on the `DefaultTimestampFormat` class which is based on the Java `DateFormat`.

In Line 15, the next execution time is computed. To this end, `ONE_DAY` is added to the current time in `now`. `ONE_DAY` is a `Duration` constant defined in Line 4. Note that adding a `Duration` to a `Timestamp` does not actually modify the `Timestamp`. All value types in Aimpulse Spectrum are immutable like the Java `String`. Therefore, the result of the `add()` method is a new `Timestamp` instance, named `nextExecution` in this example.

Listing 5.2 shows the output of this timer example. Starting in 1970, the behavior prints the date of each day. To recapitulate, the platform runs in its discrete event simulation mode throughout this tutorial. Every day, the behavior prints the date of the day. Afterwards, it requests its next scheduling for the next day. Following this request, the platform can advance time to the next day because there are no other behaviors and therefore no other events to be processed. This means that time advancement in this example simulation is significantly faster than in real time.

Listing 5.2: Console output of the timer example (first three lines)

```
The time is now 1970-01-01T00:00:00.000Z
The time is now 1970-01-02T00:00:00.000Z
The time is now 1970-01-03T00:00:00.000Z
[...]
```

In real-world operation, there is no simulation time. Therefore, it might seem acceptable to simply call `System.currentTimeMillis()` then. However, even in operation mode, it is strongly discouraged to use that method. Instead, programmers should always access the `currentTime()` method of Spectrum to ensure that their implementation is seamlessly exchangeable between simulation and operation mode.

The intention of the previous textbook example is to motivate why programmers should always use the `currentTime()` method. Actually, in this particular example there exists an even shorter notation that does not even require calling the `currentTime()` method. In Listing 5.3, the current time is only used for the console output. The next scheduling is requested in Line 6 by calling the `block()` directly with the `ONE_DAY` constant. Internally, the `BasicBehavior` then adds this `Duration` to the current `Timestamp`.

Listing 5.3: Implementation of the `AlternativeTimerBehavior`

```
1   @Override
2   public void action() {
3     Timestamp now = currentTime();
4     String formatted = ISO8601_EXTENDED_COMPLETE.format(now);
5     System.out.println("The␣time␣is␣now␣" + formatted);
6     block(ONE_DAY);
7   }
```

This example shows the advantage of value types again. If timestamp and duration values were simply represented in the `long` data type, one could add two timestamps by accident. Doing so, usually has no meaningful interpretation. With dedicated methods, value types effectively ensure that only values of the `Duration` type can be added to `Timestamp` instances.

6 Composite Agent Behavior

Sections 3 and 4 have already introduced the implementation of basic and repeated agent behavior. These individual parts of agent behavior can be composed to more complex behavior that goes beyond repeating the same task. This section describes how individual behavior instances can be combined to parallel and sequential behavior.

Listing 6.1 extends the `RepeatedBehavior` from Listing 4.3. Instead of a hard-wired text, the constructor of the new `PrintBehavior` takes the text to be written to the console as a constructor parameter. Instances of the behavior will be executed three times at most. This allows observing different behavior scheduling approaches based on the console output of the respective behaviors.

Listing 6.1: Implementation of the `PrintBehavior`

```
1  public class PrintBehavior extends BasicBehavior {
2
3    private static final int NUM_LOOPS = 3;
4    private int count = 0;
5    private String text;
6
7    public PrintBehavior(BehaviorController controller, String text) {
8      super(controller);
9      this.text = text;
10   }
11
12   @Override
13   public void action() {
14     System.out.println(text);
15     if (++count < NUM_LOOPS) {
16       reschedule();
17     }
18   }
19 }
```

Listing 6.2 gives an example how multiple behaviors can be executed in parallel. The differences to Listing 3.2 can be found in Lines 5 to 8. In Line 5, a `ParallelBehavior` is instantiated. It allows adding multiple sub-behaviors to be executed in parallel. Firstly, a behavior writing `Hello` to the console is added in Line 6. Secondly, a behavior writing `World` is added in Line 7. Finally, the `ParallelBehavior` itself is added to the agent in Line 8.

The output of this example is shown in Listing 6.3. As expected, the outputs of `Hello` and `World` are actually alternating.

Note that executing behaviors in parallel does not necessarily mean that they are executed concurrently. Moreover, note that sub-behaviors of a parallel behavior are not necessarily scheduled alternatingly. The `ParallelBehavior` is executed whenever at least one of its children demands execution. It then schedules all sub-behaviors that demand execution. This means, that programmers can easily construct non-alternating behaviors. For instance, one child behavior might be executed every 10 seconds, another every 20 seconds. So, *parallel* primarily means that the sub-behaviors do not depend on each other.

Listing 6.4 gives an example how multiple behaviors can be executed in sequence. In contrast to parallel execution, sequential behaviors depend on each other. A behavior is not executed

Listing 6.2: Implementation of the `ParallelExecution` example

```
1   public static void main(String[] arguments) {
2     Platform platform = new DefaultPlatform();
3     platform.setup(new DiscreteEventSimulation(START));
4     Agent agent = new Agent();
5     ParallelBehavior parallel = new ParallelBehavior(agent);
6     parallel.addSubBehavior(new PrintBehavior(agent, "Hello"));
7     parallel.addSubBehavior(new PrintBehavior(agent, "World"));
8     agent.addBehavior(parallel);
9     platform.addAgent(agent, NAME);
10    platform.start();
11  }
```

Listing 6.3: Console output of the `ParallelExecution` example

```
Hello
World
Hello
World
Hello
World
```

before its predecessor is finished. The differences to Listing 6.2 can be found in Lines 5 to 8. Here, a `SequentialBehavior` is used instead of a `ParallelBehavior`.

Listing 6.4: Implementation of the `SequentialExecution` example

```
1   public static void main(String[] arguments) {
2     Platform platform = new DefaultPlatform();
3     platform.setup(new DiscreteEventSimulation(START));
4     Agent agent = new Agent();
5     SequentialBehavior sequential = new SequentialBehavior(agent);
6     sequential.addSubBehavior(new PrintBehavior(agent, "Hello"));
7     sequential.addSubBehavior(new PrintBehavior(agent, "World"));
8     agent.addBehavior(sequential);
9     platform.addAgent(agent, NAME);
10    platform.start();
11  }
```

The output of this example is shown in Listing 6.5. As expected, at first, `Hello` is written three times. Afterwards, `World` is written three times. This means that actually the first behavior is executed until it is finished before the second behavior starts execution.

As a consequence of this dependency, the second behavior would never be executed if the first behavior were the one from Listing 4.2 (which is executed forever).

UML state machine diagrams of the child scheduling of both the `ParallelBehavior` and the `SequentialBehavior` can be found in the Aimpulse Spectrum implementation specification. Furthermore, there is a `FiniteStateMachineBehavior` that allows defining transitions between its child behaviors based on their outcome.

Listing 6.5: Console output of the `SequentialExecution` example

```
Hello
Hello
Hello
World
World
World
```

All composite behaviors can be combined with each other. Hence, it is possible to use any of them inside of another. Actually, already the behavior-based `Agent` itself internally contains a `ParallelBehavior`. Therefore, multiple `Behavior` instances can be added to the agent using the `addBehavior()` method.

7 Message Exchange

Software agents seldomly act in isolation. By contrast, an important property of multiagent systems is that agents interact. In order to coordinate each other, agents can exchange messages. The message structure of Aimpulse Spectrum follows the FIPA standards[1].

FIPA

The Foundation for Intelligent Physical Agents (FIPA) is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. In particular, FIPA provides standards for agent platforms with messaging and discovery services as well as for agent interaction protocols.

The example in this section is about an agent sending a message to another one. Message sending and receiving is implemented by means of `Behavior` classes. The agent initiating a communication will be referred to as the initiator. The responding agent will be referred to as the responder.

Listing 7.1 shows the implementation of the `InitiatorBehavior`. The constructor of this behavior takes an `AgentIdentifier` as an additional parameter in Line 5. This is the name of the message responder.

In the `action()` method in Line 13, the message is constructed. The `aMessage()` method is a static import from the `MessageBuilder` class. It creates a builder for a message of the specified type, a `REQUEST` message in this case. The `REQUEST` constant is itself a static import from the `Performatives` interface. The `Performatives` interface provides all message performatives defined by the FIPA. The `MessageBuilder` can be decorated with all available message properties. In this example, the `to()` method is used to define the receiver and the `containing()` method to define the message content. The Aimpulse Spectrum implementation specification gives a more detailed overview of all message properties.

Once all properties are specified, the `build()` method creates the actual message. So, the `MessageBuilder` and its `aMessage()` method provide a convenient shorthand notation

[1] `http://www.fipa.org/`

Listing 7.1: Implementation of the `InitiatorBehavior`

```
1  public class InitiatorBehavior extends BasicBehavior {
2
3    private AgentIdentifier responder;
4
5    public InitiatorBehavior(BehaviorController controller,
6        AgentIdentifier responder) {
7      super(controller);
8      this.responder = responder;
9    }
10
11   @Override
12   public void action() {
13     ModifiableMessage message = aMessage(REQUEST).to(responder)
14         .containing("Could␣you␣do␣me␣a␣favor?").build();
15     BehaviorController controller = getController();
16     controller.send(message);
17   }
18 }
```

Performative

A performative specifies a message with respect to its intention and purpose, called communicative act. For instance, the `inform` performative denotes that the message sender believes in the sent proposition and wants the receiver to believe in it to. Performatives in agent communication are defined by FIPA. In Aimpulse Spectrum, these performatives are given as typed constants in the the `Performatives` interface.

for creating messages. The resulting message is an instance of the `ModifiableMessage` interface. Therefore, all properties can still be modified afterwards.

In Line 15, a reference to the `BehaviorController` is requested. This is provided by the `getController()` method of the `BasicBehavior` class. The controller is required to actually send the message in Line 16.

Listing 7.2 shows the implementation of the `ResponderBehavior`. The `action()` method of the `ResponderBehavior` starts with requesting its `BehaviorController` in Line 9.

In Line 10, it is checked whether the controller has received any messages. If this is not already the case, the `block()` method is called and the execution of the `action()` method is temporarily ended with the `return` statement. Calling the `block()` statement without an additional parameter blocks the execution of the respective behavior until a new message arrives.

As soon as a message arrives, the `action()` method is executed again. Obviously, in this case the controller has at least a message. So, the behavior can receive it by calling the `receive()` method in Line 14. The received message implements the `Message` interface. This means that all properties can be accessed. However, the `ModifiableMessage` interface is no longer implemented so that the properties can no longer be modified. Finally, the received message is written to the console in Line 15.

Listing 7.2: Implementation of the `ResponderBehavior`

```
1  public class ResponderBehavior extends BasicBehavior {
2
3    public ResponderBehavior(BehaviorController controller) {
4      super(controller);
5    }
6
7    @Override
8    public void action() {
9      BehaviorController controller = getController();
10     if (!controller.hasMessages()) {
11       block();
12       return;
13     }
14     Message message = controller.receive();
15     System.out.println(message);
16   }
17 }
```

Listing 7.3 brings it altogether and builds a multiagent system with an initiating and a responding agent. In Line 10, the responding agent is constructed and a `ResponderBehavior` is added in Line 11. Subsequently, the agent itself is added to the platform in Line 12.

Listing 7.3: Implementation of the `Messaging` example

```
1  public class Messaging {
2
3    private static final Timestamp START = Timestamp.fromMillisecondsSince1970(0);
4    private static final String INITIATOR = "initiator";
5    private static final String RESPONDER = "responder";
6
7    public static void main(String[] arguments) {
8      Platform platform = new DefaultPlatform();
9      platform.setup(new DiscreteEventSimulation(START));
10     Agent responder = new Agent();
11     responder.addBehavior(new ResponderBehavior(responder));
12     AgentIdentifier address = platform.addAgent(responder, RESPONDER);
13     Agent initiator = new Agent();
14     initiator.addBehavior(new InitiatorBehavior(initiator, address));
15     platform.addAgent(initiator, INITIATOR);
16     platform.start();
17   }
18 }
```

In the previous examples, the return value of the `addAgent()` has simply been ignored. In this example, it is kept in the `address` variable. It is an `AgentIdentifier` instance that can be used to address the respective agent. Usually, this `AgentIdentifier` will simply encapsulate the `String` parameter of the `addAgent()` method (the value `responder` of the `RESPONDER` constant in this example). However, this parameter is just a proposal to the platform. In particular, agent names must be unique throughout the multiagent system. Therefore, a serial number might be added to the proposed name if it is already in use.

In Line 13, the initiating agent is constructed and a `InitiatorBehavior` is added in Line 14. The `address` of the responding agent is used as the second parameter of the `InitiatorBehavior`. Finally, also the initiating agent itself is added to the platform in Line 15.

Listing 7.4 shows the output of this example. It is the message that the responding agent prints to the console. The output follows the XML representation specified by the FIPA. It contains information about the performative, sender and receiver, and the actual content of the message. If additional parameters were defined, they would also be contained in the XML output.

Listing 7.4: Console output of the `Messaging` example

```
<fipa-message act="request">
<sender>
<agent-identifier>
<name id="initiator"/>
</agent-identifier>
</sender>
<receiver>
<agent-identifier>
<name id="responder"/>
</agent-identifier>
</receiver>
<content>Could you do me a favor?</content>
</fipa-message>
```

An `AlternativeResponderBehavior` is implemented in Listing 7.5. Differences to the previous one can be found in Lines 17 and 21. Both the `hasMessages()` and the `receive()` methods are called with a `MessagePattern` parameter. Without this parameter, the methods check and return arbitrary messages. With this parameter, they only consider messages matching the respective pattern.

The `PATTERN` itself is implemented in an anonymous class in Lines 3 to 8. The particular implementation here matches only messages with a `REQUEST` performative. Using message patterns is useful whenever multiple messages are sent. They enable to delegate messages to the respective behaviors handling them. Furthermore, sequences of messages (as in interaction protocols) can be dealt with conveniently with message patterns.

In Line 23, there is another extension in the `AlternativeResponderBehavior`. A reply to the original message is created. This is done using the static `aReply()` method. Like the `aMessage()` method, `aReply()` is a static import from the `MessageBuilder` class. In addition to the original message, it takes the performative of the reply as an argument (the statically imported `REFUSE` performative in this example). Afterwards, the same decorators as for the `aMessage()` method can be applied. The `aReply()` method automatically sets all properties related to conversation control. In this example, it sets the sender of the original message as the receiver of the reply. The reply is then sent in Line 24.

8 Interaction Protocols

The example in Section 7 has already introduced message exchange. However, communication becomes increasingly complex with an increasing number of messages exchanged. Interaction

Listing 7.5: Implementation of the `AlternativeResponderBehavior`

```
1  public class AlternativeResponderBehavior extends BasicBehavior {
2
3    private static final MessagePattern PATTERN = new MessagePattern() {
4      @Override
5      public boolean covers(Message message) {
6        return REQUEST.equals(message.getPerformative());
7      }
8    };
9
10   public AlternativeResponderBehavior(BehaviorController controller) {
11     super(controller);
12   }
13
14   @Override
15   public void action() {
16     BehaviorController controller = getController();
17     if (!controller.hasMessages(PATTERN)) {
18       block();
19       return;
20     }
21     Message message = controller.receive(PATTERN);
22     System.out.println(message);
23     ModifiableMessage reply = aReply(message, REFUSE).build();
24     controller.send(reply);
25   }
26 }
```

protocols help cope with this complexity by structuring the message flow. Aimpulse Spectrum is shipped with behavior-based implementations of the most popular interaction protocols specified by the FIPA. This section reimplements the example from Section 7 based on the FIPA Request Interaction Protocol.

Listing 8.1 shows the implementation of the `InitiatorAgent`. Note the difference to the previous examples that only implemented behaviors. In general, the source code can be better organized if the agent setup is moved to the agent class. This particularly holds if there is more than one behavior or if the behaviors require further configuration.

In Line 5, the agent is instantiated with the `AgentIdentifier` of the responder. Further setup is done in the `setup()` method instead of the constructor. The `setup()` is called once the agent is executed for the first time. At that time, the agent is already configured for execution. By contrast, important parameters such as the name of the agent might not be available at construction time.

In Line 11, the request message is created. Afterwards, it is used as the second constructor parameter for a `RequestInitiator` instance in Line 13. The first parameter is a reference to the agent itself which acts as the `BehaviorController`. The `RequestInitiator` is a behavior that is responsible for the initiator role in the FIPA Request Protocol. It sends the request and handles incoming responses. To be actually executed, the `initiator` behavior is added to the agent in Line 19.

With the `RequestInitiator`, the handling of response messages can be implemented in two ways. By means of call-back methods or by registering additional behaviors. In this example, only the `handleRefuse()` method is implemented in Line 15 in order to handle

Listing 8.1: Implementation of the `InitiatorAgent`

```
1  public class InitiatorAgent extends Agent {
2
3    private AgentIdentifier responder;
4
5    public InitiatorAgent(AgentIdentifier responder) {
6      this.responder = responder;
7    }
8
9    @Override
10   public void setup() {
11     ModifiableMessage request = aMessage(REQUEST).to(responder)
12         .containing("Could␣you␣do␣me␣a␣favor?").build();
13     Behavior initiator = new RequestInitiator(this, request) {
14       @Override
15       public void handleRefuse(Message refuse) {
16         System.out.println(refuse);
17       }
18     };
19     addBehavior(initiator);
20   }
21 }
```

incoming refuse messages. More specifically, the respective message is simply written to the console. The Aimpulse Spectrum implementation specification gives a more detailed description of the protocol including an overview of all available call-back methods. Furthermore, it describes how to register additional behaviors instead of call-back methods.

Listing 8.2 shows the implementation of the `ResponderAgent`. In the `setup()` method in Line 5, a new `RequestResponder` behavior is instantiated. It is responsible for the initiator role in the FIPA Request Protocol. The `responder` behavior is added to the agent in Line 12.

Listing 8.2: Implementation of the `ResponderAgent`

```
1  public class ResponderAgent extends Agent {
2
3    @Override
4    public void setup() {
5      Behavior responder = new RequestResponder(this) {
6        @Override
7        public ModifiableMessage handleRequest(Message request)
8            throws NotUnderstoodException, RefuseException {
9          throw new RefuseException();
10       }
11     };
12     addBehavior(responder);
13   }
14 }
```

The `RequestResponder` supports two call-back methods: `handleRequest()` to agree or refuse requests and `prepareResultNotification()` to create the actual response. As in the previous example, the responder simply refuses the request. The simplest way to do so is to simply throw a `RefuseException` as in Line 9. The implementation of the `RequestRe-`

sponder internally converts this exception to the respective response. Positive responses can be created by returning messages created with the previously introduced aReply() method.

Listing 8.3 brings it altogether and builds a multiagent system with an initiating and a responding agent. First, the ResponderAgent is instantiated and added to the agent platform in Line 10. Like in the previous example, its address is kept and passed to the InitiatorAgent that is instantiated and added in Line 12.

Listing 8.3: Implementation of the Interaction example

```
1  public class Interaction {
2
3    private static final Timestamp START = Timestamp.fromMillisecondsSince1970(0);
4    private static final String INITIATOR = "initiator";
5    private static final String RESPONDER = "responder";
6
7    public static void main(String[] arguments) {
8      Platform platform = new FIPAPlatform();
9      platform.setup(new DiscreteEventSimulation(START));
10     AgentIdentifier address =
11         platform.addAgent(new ResponderAgent(), RESPONDER);
12     platform.addAgent(new InitiatorAgent(address), INITIATOR);
13     platform.start();
14   }
15 }
```

The console output of this example is given in Listing 8.4. It shows the response message received by the initiator. Note the additional parameters for conversation control that have been added automatically by the underlying behavior implementations. The protocol identifier and the conversation-id help match responses to the original request. Furthermore, each request has a reply-with parameter that helps distinguish responses if multiple responders are involved. Here, the responder references that value in the in-reply-to parameter.

Listing 8.4: Console output of the Interaction example

```
<fipa-message act="refuse">
<sender>
<agent-identifier>
<name id="responder"/>
</agent-identifier>
</sender>
<receiver>
<agent-identifier>
<name id="initiator"/>
</agent-identifier>
</receiver>
<protocol>fipa-request</protocol>
<conversation-id>initiator-0-0</conversation-id>
<in-reply-to>initiator-0-1</in-reply-to>
</fipa-message>
```

9 Agent Discovery

In the previous interaction examples in Sections 7 and 8, the communication partner is already known in advance. The address of the responder is passed to the initiator during its instantiation. In real applications, however, communications partners are usually not known before. Multiagent systems are open so that agents may join and leave the system at any time. Therefore, agents must be able to discover other agents during runtime. This demand is addressed by the directory service, the yellow pages of the multiagent system. This section presents an example with two agents that do not know each other initially. One is a provider of a particular service. The other is a consumer that requests the respective service.

Listing 9.1 shows the implementation of the `ProviderAgent`. This agent offers a service that it wants to announce via the directory service. This is prepared in its `setup()` method. In Line 5, a respective `ServiceDescription` is created by means of the `aService()` method. It is a static import from the `ServiceDescriptionBuilder` class. The `ServiceDescriptionBuilder` can be decorated with all available service description properties. In this example, the `named()` method is used to define the name of the service. In this example, the service name is defined in the `ROUTING` constant of the `Discovery` class. The Aimpulse Spectrum implementation specification gives a more detailed overview of all service description properties. Once all properties are specified, the `build()` method creates the actual service description.

Listing 9.1: Implementation of the `ProviderAgent`

```
1  public class ProviderAgent extends Agent {
2
3    @Override
4    public void setup() {
5      ServiceDescription routing = aService().named(Discovery.ROUTING).build();
6      AgentIdentifier name = getAddress();
7      AgentDescription description = anAgent().named(name)
8          .offering(routing).build();
9      Behavior register = new RegisterInitiator(this, description);
10     addBehavior(register);
11   }
12 }
```

In Line 7, an `AgentDescription` is created by means of the `anAgent()` method. It is a static import from the `AgentServiceDescriptionBuilder` class. The `AgentServiceDescriptionBuilder` can be decorated with all available agent description properties. In this example, the `named()` method is used to define the name of the agent. This has been retrieved before using the `getAddress()` method that returns the name of the agent. Furthermore, the `offering()` method is used to add the offered service to the agent description. The Aimpulse Spectrum implementation specification gives a more detailed overview of all agent description properties. Once all properties are specified, the `build()` method creates the actual agent description.

In Line 9, a `RegisterInitiator` is instantiated with the agent description. It is a `Behavior` that is responsible for registering the description with the directory. Similar behaviors exist for updating and for deregistering agent descriptions. Finally, the behavior is added to the agent in Line 10 in order to be actually executed.

Listing 9.2 shows the implementation of the other participant in this example: the `ConsumerAgent`. This agent demands the service offered by the other one. However, it does not know the provider initially. Therefore, it has to search for a suitable provider. This is prepared in its `setup()` method. In Line 5, a description of the desired service is created. This is the same procedure as done by the provider.

Listing 9.2: Implementation of the `ConsumerAgent`

```
1  public class ConsumerAgent extends Agent {
2
3    @Override
4    public void setup() {
5      ServiceDescription routing = aService().named(Discovery.ROUTING).build();
6      AgentDescription description = anAgent().offering(routing).build();
7      SearchInitiator search = new SearchInitiator(this, description) {
8        @Override
9        public void handleSearchResults(AgentDescription[] results) {
10         if (results.length == 0) {
11           System.out.println("No␣matching␣agents␣found.");
12           return;
13         }
14         System.out.println("Matching␣agents␣found:");
15         for (AgentDescription result : results) {
16           System.out.println(result.getName());
17         }
18       }
19     };
20     addBehavior(search);
21   }
22 }
```

In Line 6, a description of the required agent is created. In this case, the name does not play any role. Instead, it suffices that the respective agent offers the desired service. Therefore, the `ServiceDescription` is used to create the `AgentDescription`.

In Line 7, a `SearchInitiator` is instantiated with the agent description. It is a `Behavior` that is responsible for searching agents registered with the directory. The behavior is added to the agent in Line 20 in order to be actually executed.

The `SearchInitiator` has a `handleSearchResults()` call-back method that can be used to process the search results. In Line 10, it is checked whether there are actually search results. If no agents have been found, a respective notification is written to the console and the method is ended with the `return` statement. If there are search results, the names of the agents found are written to the console.

Listing 9.3 brings it altogether and builds a multiagent system with a provider and a consumer agent. The `ROUTING` constant that is the `ServiceIdentifier` jointly used by the provider and consumer is defined in Line 3.

In Line 12, a `FIPAPlatform` is instantiated. Note the difference to the previous examples when a `DefaultPlatform` was used. On the `FIPAPlatform`, there is already a directory facilitator agent by default.

Listing 9.3: Implementation of the `Discovery` example

```
1  public class Discovery {
2
3    public static final ServiceIdentifier ROUTING =
4        new ServiceIdentifier("routing");
5
6    private static final Timestamp START = Timestamp.fromMillisecondsSince1970(0);
7    private static final Timestamp LATER = Timestamp.fromMillisecondsSince1970(10);
8    private static final String PROVIDER = "provider";
9    private static final String CONSUMER = "consumer";
10
11   public static void main(String[] arguments) {
12     Platform platform = new FIPAPlatform();
13     platform.setup(new DiscreteEventSimulation(START));
14     platform.addAgent(new ProviderAgent(), PROVIDER);
15     platform.addAgent(new ConsumerAgent(), CONSUMER, LATER);
16     platform.start();
17   }
18 }
```

In Line 14, a `ProviderAgent` is added to the platform. In Line 15, a `ConsumerAgent` is added to the platform. Note the additional parameter `LATER` when adding the `ConsumerAgent`. Without this parameter, agents are immediately added to the platform. With the parameter, agents are not added before the respective `Timestamp`. In the case of the `LATER` constant, it is a `Timestamp` ten milliseconds after the start of the platform. The deferred adding of the agent is done intentionally here. The provider needs some time to register with the directory because message exchange consumes time. Consequently, it must be ensured that the consumer does not start searching for the provider before the provider is registered.

The console output in Listing 9.4 shows that the consumer actually finds the provider agent in the directory.

Listing 9.4: Console output of the `Discovery` example

```
Matching agents found:
provider
```