Chapter 2
Getting Started with GORITE

Goal Oriented Teams (GORITE) is a Java framework for the implementation of goal oriented process models in a team oriented paradigm (Rönnquist, 2012). Both the design and implementation of GORITE is representative of industry best practice. The framework is also the subject of ongoing maintenance and enhancement – the current release (June 2012) is v9RC04. Given its primary design objective, GORITE is provided as a Java class library and there is currently no graphical development environment. However, in contrast to conventional BDI frameworks, there is no separate plan language – all development is in Java. Consequently, developers can continue to use their favourite IDE.

From a programming perspective, the GORITE API conforms, where practical, to JDK1.4. The reason for this is to make porting to new platforms more straightforward. When developing applications, developers are of course free to use features from the current version of Java. The GORITE API is documented using JavaDoc, and excerpts are available in this book as Appendix B. There is no separate user documentation – this book represents an attempt to fill that void.

With the exception of the examples presented in this chapter, the examples in this book are incomplete in that package statements, import statements and constant definitions are omitted. However, working code for all the examples is available from the GORITE website (http://www.intendico.com/GORITE). With respect to constants, we use the convention that their names are specified in uppercase. Exception handling is included when necessary, either by throwing or catching the exception. However, in the latter case, no behaviour is specified for when the exception is caught. Also note that output is sent to System.err, not System.out. The reason for this is that System.err is unbuffered, so output appears immediately it is initiated.

2.1 Hello World

“Hello World” represents, since its introduction in (Kernighan and Ritchie, 1978) the archetype for a first program when presenting a new programming language. We will not break with tradition here, but what we will do is introduce, in addition to a canonical version, a number of variants. The purpose of these variants is to illustrate, at an introductory level, how the GORITE execution model operates and how the key BDI concepts of beliefs, desires and intentions are represented. The rationale for this is that the representational structure and the execution model together provide a convenient mechanism for characterizing BDI frameworks.
2.1.1 Version 1: Goal Execution

In this example we will create an alien performer (a.k.a. agent) that can achieve a Greetings goal by printing the message “Hello World”. The initial application consists of three source files – Alien.java, Greetings.java and Main.java. The code for the Alien class is shown below:

**Alien.java:**

```java
package ch2.greetings.v1a;
import com.intendico.gorite.*;

public class Alien extends Performer {

    // constructor
    Alien(String name) {
        super(name);
    }

    // A method to initiate a greeting goal
    public boolean greet() {
        // Greet the people
        return performGoal(new Greetings(), "Greetings", new Data());
    }
}
```

This code resides in a file called Alien.java. The alien will be created from a main() method contained in Main.java. The alien’s greet() method will also be invoked from main(). The Greetings goal is specified in Greetings.java. Listings for Main.java and Greetings.java are presented below.

Note that in the code above, the Alien class extends com.intendico.gorite.Performer. The Performer class has a constructor which has a single parameter – the name of the performer. It also makes available the performGoal() method that is employed within the subclass for goal execution. The first parameter for performGoal() is the goal instance to be executed. The second parameter is used in execution tracing – it provides the “head” for the goal/process model that is being executed. In this case, the process model is a single goal. The final parameter holds the data context for the goal execution. If the goal execution requires access to any data elements for its completion, these elements would be added to the data context. In this example, no application data is required and therefore no data elements are added to the data context.

Goal execution is not performed directly by the performer. Rather, performers delegate execution to one or more instances of the Executor class. An executor traverses a process model, executing sub-goals as dictated by the model structure.

---

1 Tracing is discussed in Appendix A.