*"HLA Development Kit"*

*A brief guide*

# *Table of contents*

**1 Introduction ..... 4**

**2 The HLA Development Kit Framework ..... 4**

**2.1 Overview ..... 5**

**3 Architecture of the DKF ..... 6**

**3.1 DKF packages ..... 8**

3.1.1 The dkf.core package ..... 10

3.1.1.1 The dkf.core.observer package ..... 11

3.1.1 The dkf.config package ..... 12

3.1.2 The dkf.logging package ..... 13

3.1.3 The dkf.exception package ..... 13

3.1.4 The dkf.coder package ..... 14

3.1.5 The dkf.utility package ..... 15

3.1.6 The dkf.model package ..... 16

3.1.6.1 The dkf.model.parser package ..... 17

3.1.6.2 The dkf.model.object package ..... 17

3.1.6.3 The dkf.model.interaction package ..... 19

3.1.7 The dkf.time package ..... 20

**4 The Behavioral Model of a DKF-based Federate ..... 21**

**4.1 Concurrency aspects ..... 23**

**5 Exploiting the HLA Development Kit ..... 24**

**5.1 The SEE HLA Starter Kit ..... 24**

**5.2 The development process of a DKF/SEE-SKF based Federate ..... 25**

**5.3 Using the DKF/SEE-SKF framework: the "TestFederate" ..... 26**

5.3.1 The Rover class (ObjectClass) ..... 28

5.3.2 The Interaction class (InteractionClass) ..... 29

5.3.3 The TestFederate class ..... 30
5.3.4 The TestAmbassador class ..... 33
5.3.5 The Main class ..... 33
5.3.6 The Rover FOM ..... 35
5.3.7 The Configuration file ..... 36

**Appendix A: The main services of the HLA Development Kit ..... 38**

**References ..... 39**

# 1 Introduction

This document contains an overview of the "HLA Development Kit" [5], a Java-based software development kit that aims at easing the development of HLA Federates.

The HLA Development Kit provides high level functionalities both to implement HLA Federates and manage the interactions between them and the RTI. It aims at easing the development of HLA Federates by providing the following resources: (i) a *software framework (*the *DKF)* for the development in Java of HLA Federates; (ii) a *technical documentation* that describes the DKF; (iii) a *user guide* to support developers in the use of the DKF; (iv) a set of *reference examples* of HLA Federates created by using the DKF; and, (v) *video-tutorials*, which show how to create both the structure and the behavior of a HLA Federate by using the DKF (see [1], [1] for the details).

The document is organized as follow: Section 2 provides an overview of the "HLA Development Kit Framework (DKF)"; Section 3 reports the architecture of the DKF; Section 4 describes the behavioral model of a DKF-based Federate. Finally, in Appendix B, the Java code related to an example HLA Federate developed by using the DKF is reported.

# 2 The HLA Development Kit Framework

The development of a HLA Federation with its Federates is a quite complex task and there are few training resources for developers [4]. The HLA Development Kit aims at easing the development of HLA Federates by providing the following resources: (i) a *software framework* (the *DKF*) for the development in Java of HLA Federates; (ii) a *technical documentation* that describes the DKF; (iii) a *user guide* to support developers in the use of the DKF; (iv) a set of *reference examples* of HLA

Federates created by using the DKF; and, (v) *video-tutorials*, which show how to create both the structure and the behavior of a HLA Federate by using the DKF.
In the following, the attention is focused on the DKF and, specifically, on its architecture and underlying Federate model-behavior.

## 2.1 Overview

The DKF is a general-purpose, domain-independent framework, released under the open source policy Lesser GNU Public License (LGPL), which facilitates the development of HLA Federates. Indeed, the DKF allows developers to focus on the specific aspects of their own Federates rather than dealing with the common HLA functionalities, such as: the management of the simulation time; the connection/disconnection on/from the HLA RTI; the publishing, subscribing, and updating of *ObjectClass* and *InteractionClass* elements (see [1], [3]). The DKF is designed and developed by the SMASH-Lab (System Modeling And Simulation Hub - Laboratory) of the University of Calabria (Italy) working in cooperation with the NASA JSC (Johnson Space Center), Houston (TX, USA).
The DKF is fully implemented in the Java language and is based on the following three principles: (i) *Interoperability*, DKF is fully compliant with the IEEE 1516-2010 specifications; as a consequence, it is platform-independent and can interoperate with different HLA RTI implementations (e.g. PITCH, VT/MÄK, PoRTIco, CERTI); (ii) *Portability* and *Uniformity*, DKF provides a homogeneous set of APIs that are independent from the underlying HLA RTI and Java version. In this way, developers could decide the HLA RTI and the Java run-time environment at development-time; and (iii) *Usability*, the complexity of the features provided by the DKF framework are hidden behind an intuitive set of APIs.

The design and implementation of the DKF has been centered on typical Software Engineering methods and, in particular, on an agile software development process (see Figure 1). Furthermore, it has been developed according to the concept of Object HLA, in this way, the development of HLA Federates could benefit also from the Object HLA features and functionalities provided by the Pitch Developer Studio or similar IDE.

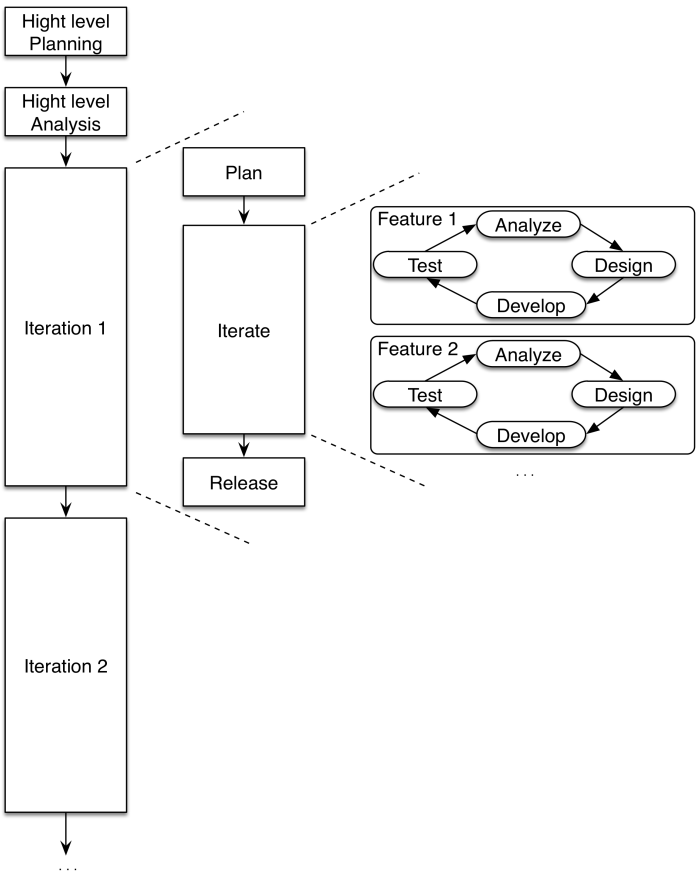


**Figure 1: The Agile software development process exploited for the DKF implementation.**

## 3 Architecture of the DKF

The DKF provides a set of Java classes that are able to handle the main and common mechanisms of a HLA Federate and that can be easily extended and

integrated with federate-specific code. As a consequence, a Federate will be composed by a set of classes provided by the DKF that are extended and/or integrated by the developer with classes that implement the specific behavior of the Federate.

The DKF provides the following main services (see Section 4 for a more complete list):

1. *Simulation Time Management:* management of the *"logical time"* of a Federate during the simulation execution;
2. *Connection Management:* handling of the phases of set-up, hold-up and close-up of the connection to the RTI (Run Time Infrastructure);
3. *Interaction Management:* handling of the notification coming from/direct to the RTI concerning the interactions with other federates.

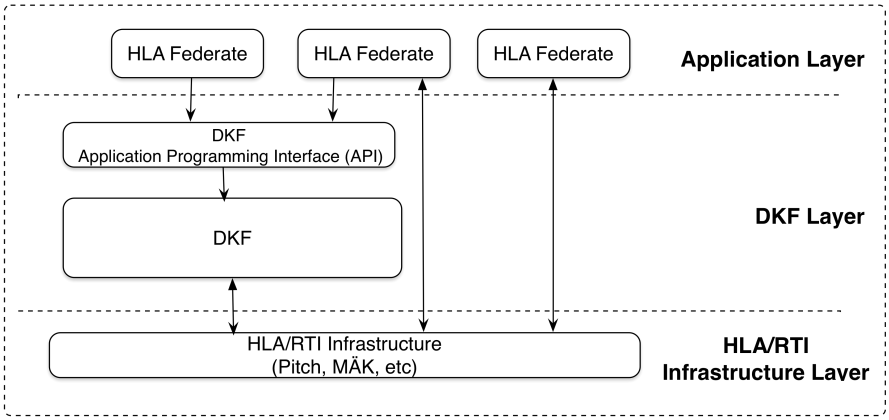


**Figure 2: The architecture of a DKF-based Federation.**

The architecture of a DKF-based Federation is composed of three main layers (see Figure 2): (i) *Application Layer*, which contains the Federates that can interact with both the DKF and the HLA RTI by using the APIs provided by them; (ii) *DKF Layer*, which represents the core of the architecture and provides a set of domain-independent APIs that are used to access the DKF capabilities; and (iii) *HLA RTI*

*Infrastructure*, which represents the RTI that host the Federation (e.g. PITCH, VT/MÄK, PoRTIco, CERTI).

## 3.1 DKF packages

The *DKF* is organized into a hierarchy of packages and sub-packages; each of which contains a set of Java classes and interfaces that implement specific functionalities; the main packages are shown in Figure 3 by using a UML package diagram.

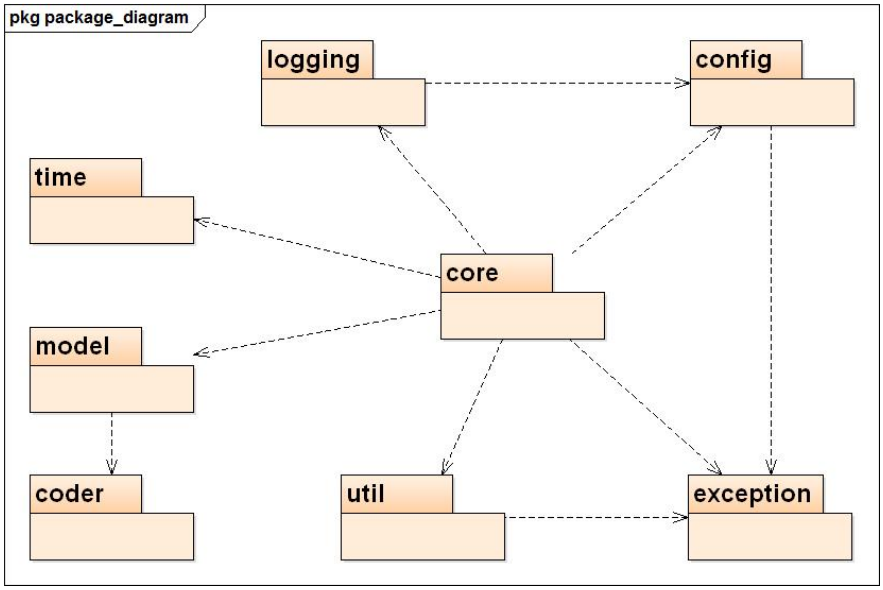


**Figure 3: DKF - UML package diagram.**

In particular, the *DKF* is composed of seven main packages that are independent both of application domains and HLA RTI implementations; in the following each of them is briefly described (please note that it is supposed that the reader is familiar with the main HLA concepts and features; if not an introductory HLA tutorial can be found at http://www.pitch.se/hlatutorial):

— **dkf.core**, implements the kernel of the DKF framework. It includes the fundamental *dkf.core.DKFAbstractFederate* and *dkf.core.*

*DKFAbstractFederateAmbassador* classes that provide the basic functionalities to manage a Federate. This package also includes a sub-packages that implementing a specific core-level service; this is:

- **dkf.core.observer,** which manages the updates concerning both *Object Attributes* and *Interactions*; this package allows a Federate to be notified whenever its *FederateAmbassador* receives an attribute/interaction update.

— **dkf.config**, which contains the collection of classes that manage the configuration parameters provided by a "*.json"* file.

— **dkf.utility**, which contains several miscellaneous utility classes, such as time standard conversions (e.g. JulianDate, RJulianDate, etc.) and Windows Firewall Check.

— **dkf.logging**, which contains a set of classes used to track down any problems or error occurred during the execution of the Federate; these information are stored into a "*dkf.log"* file.

— **dkf.exception**, which contains some definitions of exceptions that are used for handling dysfunctional events throughout the DKF framework.

— **dkf.model,** which contains some classes to facilitate publishing, subscribing and the data updating of both *ObjectClass*es and *InteractionClass*es. This package also includes the three following sub-packages that implement specific services:

- **dkf.model.object,** which defines the annotations that have to be used by the programmer to create an *ObjectClass* instance compatible with the kit;

- **dkf.model.interaction,** which defines the annotations that have to be used by the programmer to create an *InteractionClass* compatible with the kit;
- **dkf.model.parser,** the parser to inspect the structure of objects instances of the *ObjectClass* or *InteractionClass*.

— **dkf.coder,** which contains a set of classes that deal with coding and decoding of objects instances of the *ObjectClass* or *InteractionClass*.

— **dkf.time,** which contains some classes that are used to properly instantiate the simulation time, according to a FOM[1].

### 3.1.1 The dkf.core package

The architecture of the dkf.core package is shown in Figure 4 by using a UML Class Diagram; in the following its main classes are briefly described.

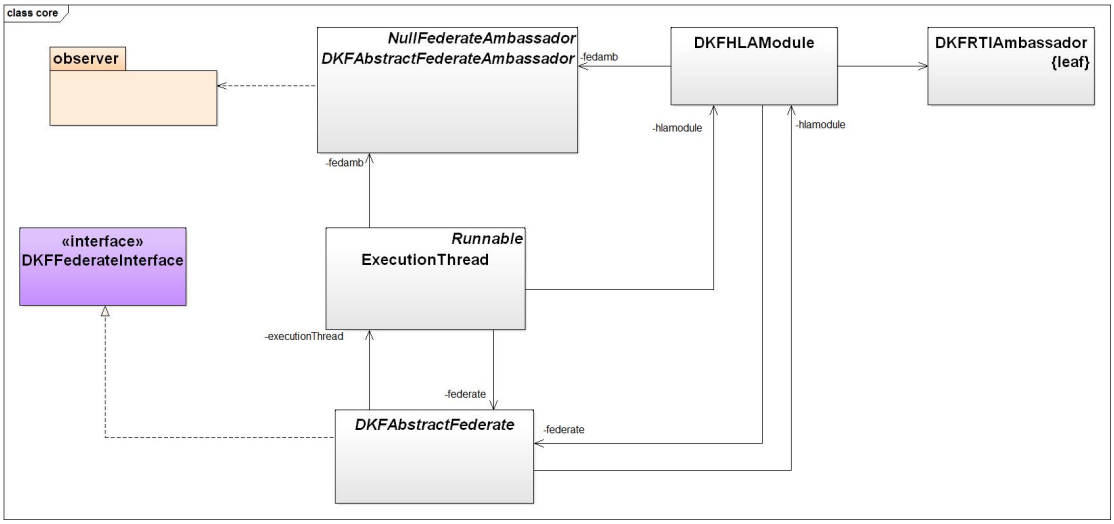


**Figure 4: The UML class diagram of the dkf.core package.**

The *DKFAbstractFederate* class manages the life cycle of a Federate; it provides functionalities to configure and connect/disconnect the Federate to/from the

[1] In this version, the DKF only provides support to the standard time representation: HLAInteger64Time and HLAfloat64Time.

Federation. In particular, the *DKFAbstractFederate* class provides a concrete Federate with the management of its life cycle (FLCM), as a consequence, a working team has only to define the specific behavior of its Federate without worrying about low-level and basic HLA functionalities since the DKF manages them (see Section 4).

The *ExecutionThread* class handles the execution of a Federate in the simulation environment.

The *DKFAbstractFederateAmbassador* class implements the methods that are called by the RTI for interacting with the Federate (RTI callback methods); along with the RTI Ambassador interface, which is used by the Federate to access the RTI services.

#### 3.1.1.1 The dkf.core.observer package

This package implements the *Observer Pattern*, in which an object class (the *Subject*), maintains a list of its dependents (the *Observers*), and automatically notifies them of any state changes.

The above-mentioned pattern has been exploited in the specific problem related to the notification of a Federate about the updates of the subscribed *ObjectClass*es and/or *Interaction*s. In this case, the *Subject* class represents, according to the patter notation, the subject component; whereas the implementation of the concrete *Observer* class is in charge of the developer.

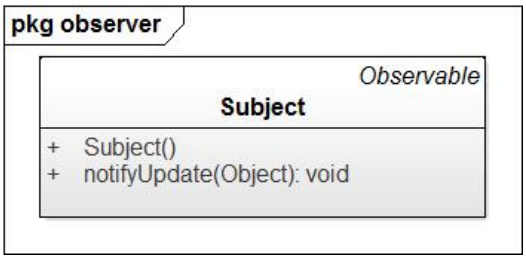


**Figure 5: The UML class diagram of the dkf.config.observer package.**

The *Subject* class notifies its *Observer*s whenever another Federate updates a subscribed object and/or interaction.

The architecture of the above described implementation of the dkf.config.observer package is shown in Figure 5 by using a UML Class Diagram.

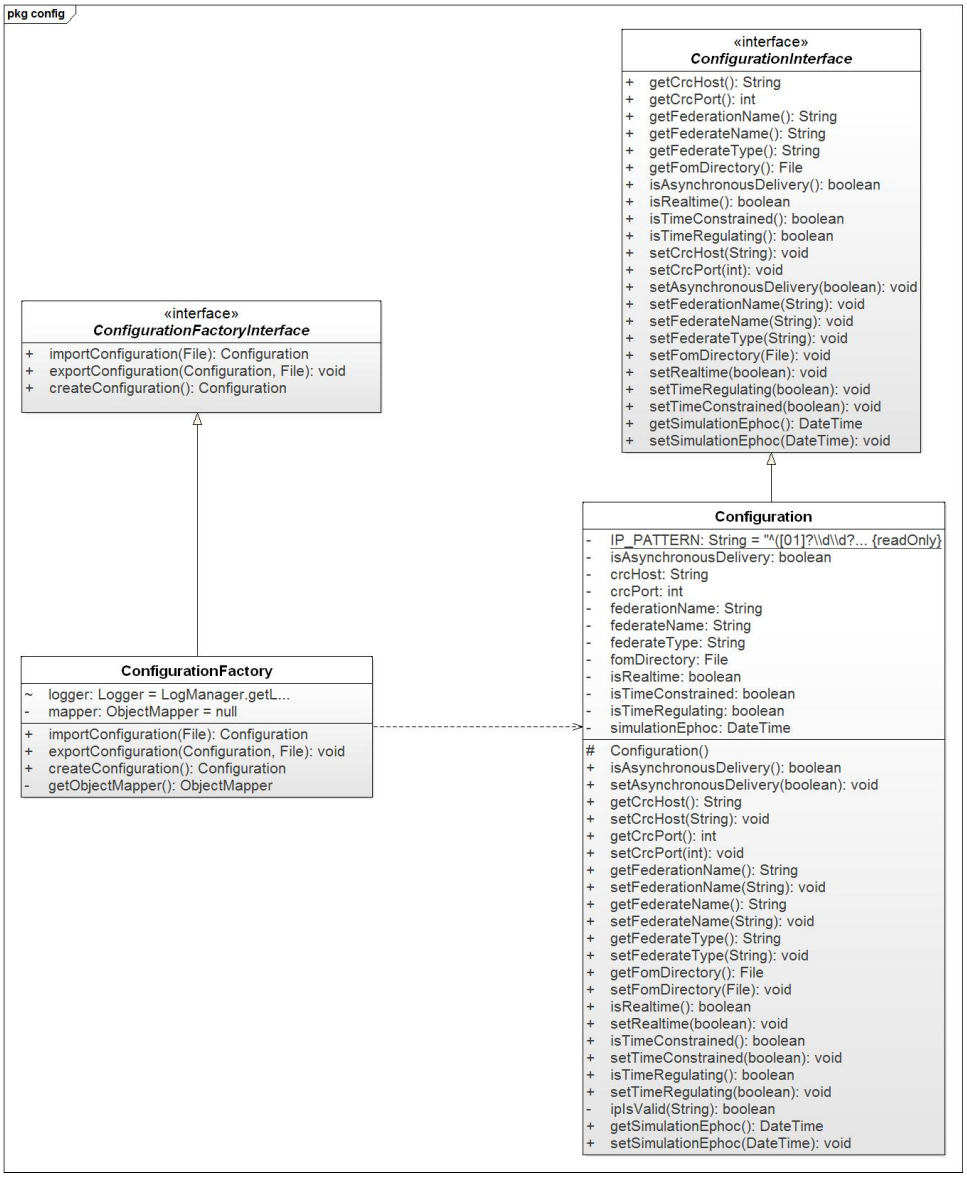


**Figure 6: The UML class diagram of the dkf.config package.**

### 3.1.1 The dkf.config package

The architecture of the dkf.config package is shown in Figure 6 by using a UML Class Diagram.

This package manages the configuration parameters of the DKF framework, which are stored in a *.json* file. The parameters concern important Federation aspects, such as the name and the type of the Federation ("federationName" and "federationType"), and the FOM directory ("fomDirectory").

The *Configuration* class manages the configuration parameters. The *ConfigurationFactory* class implements the *ConfigurationFactoryInterface* and manages the creation, loading and storing of a Configuration object.

### 3.1.2 The dkf.logging package

The architecture of the dkf.logging package is shown in Figure 7 by using a UML Class Diagram.

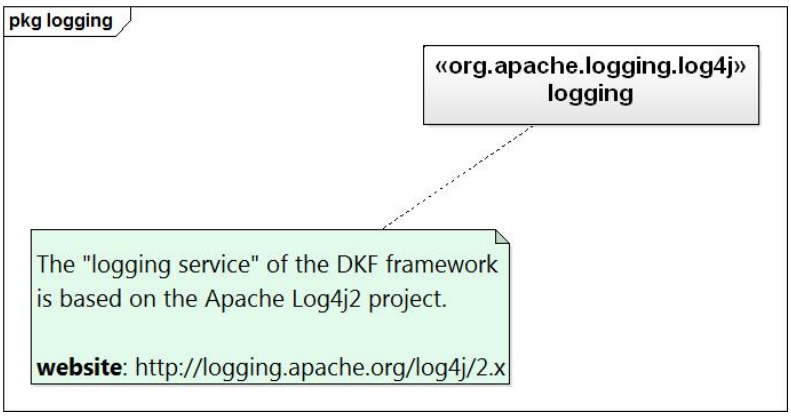


**Figure 7: The UML class diagram of the dkf.logging package.**

The DKF framework uses the *Apache Log4j2* library to manage the log events generated by the framework during the simulation execution.

### 3.1.3 The dkf.exception package

The architecture of the dkf.exception package is shown in Figure 8 by using a UML Class Diagram.

This package contains classes used to handle the unexpected situations that might occur during the simulation execution.

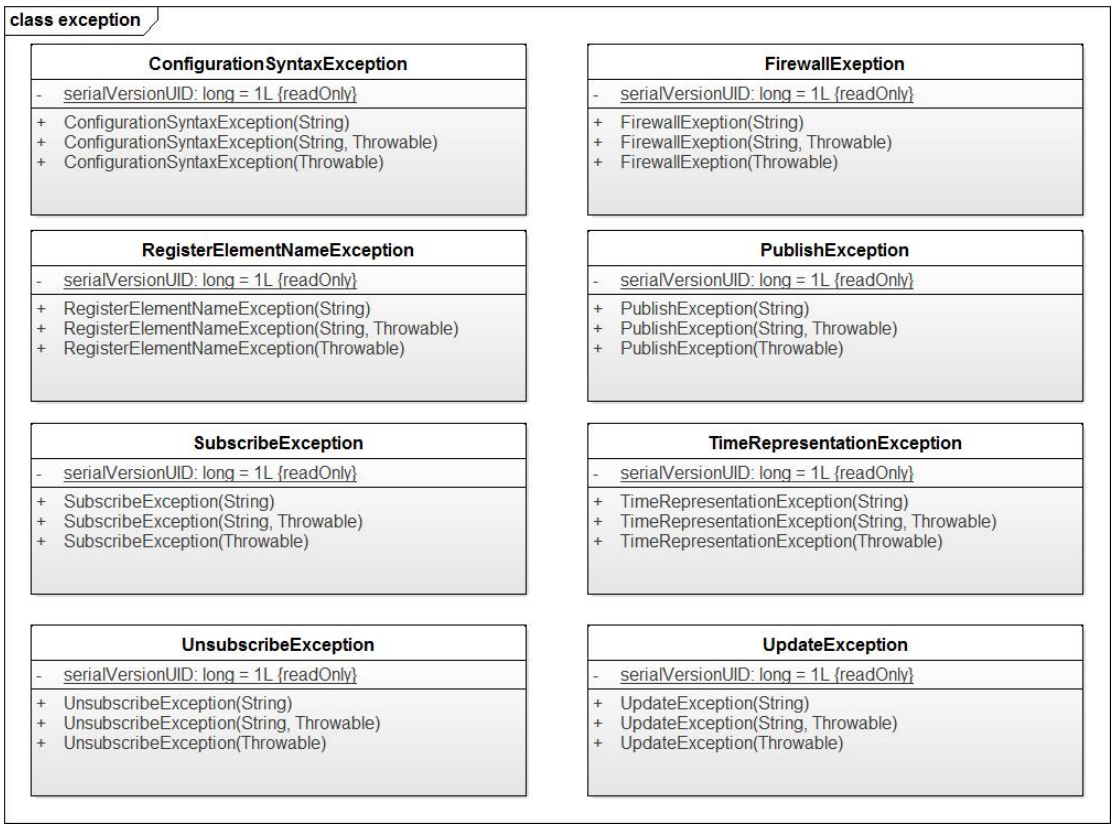


**Figure 8: The UML class diagram of the dkf.exception package.**

### 3.1.4 The dkf.coder package

The architecture of the dkf.coder package is shown in Figure 9 by using a UML Class Diagram.

This package contains classes used to encode and decode the fields of objects instances of the *ObjectClass* or *InteractionClass* during both the phases of publication and updating.

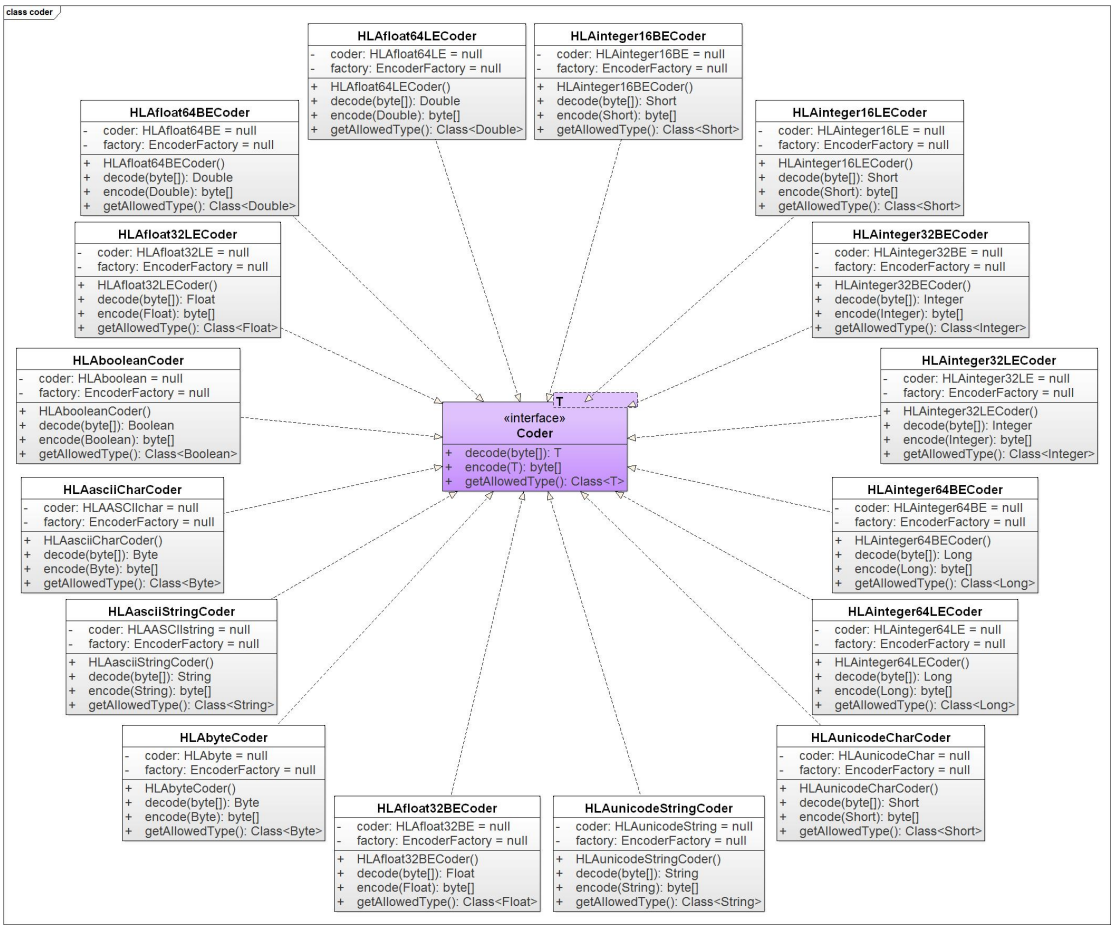


**Figure 9: The UML class diagram of the dkf.coder package.**

### 3.1.5 The dkf.utility package

The architecture of the dkf.utility package is shown in Figure 10 by using a UML Class Diagram.

This package contains several miscellaneous utility classes. In particular, the *TimeUtility* class defines the mechanisms for the time standard conversions; and, the *SystemUtility* class provides a method to check the status of the MSWindows Firewall.

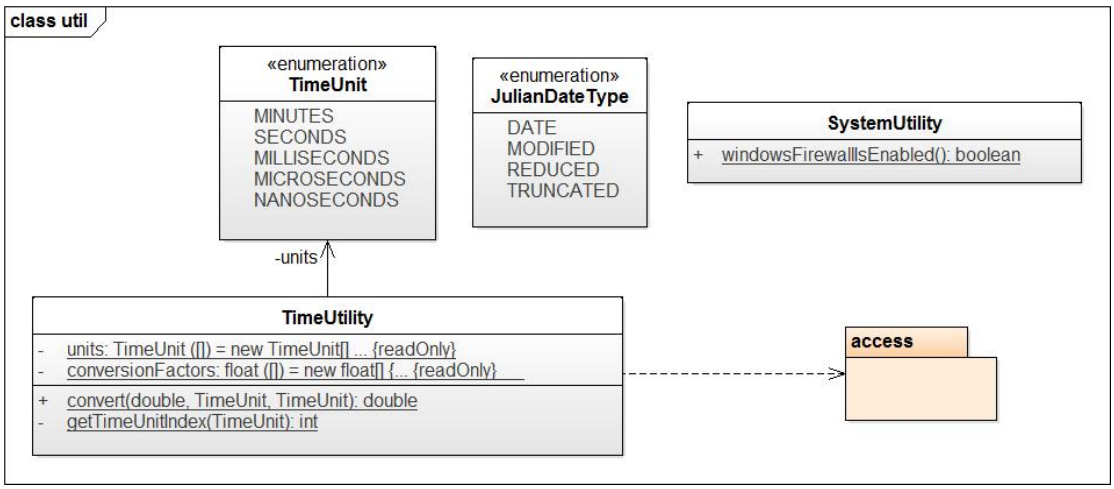


**Figure 10: The UML class diagram of the dkf.utility package.**

### 3.1.6 The dkf.model package

The architecture of the dkf.model package is shown in Figure 11 by using a UML Class Diagram.

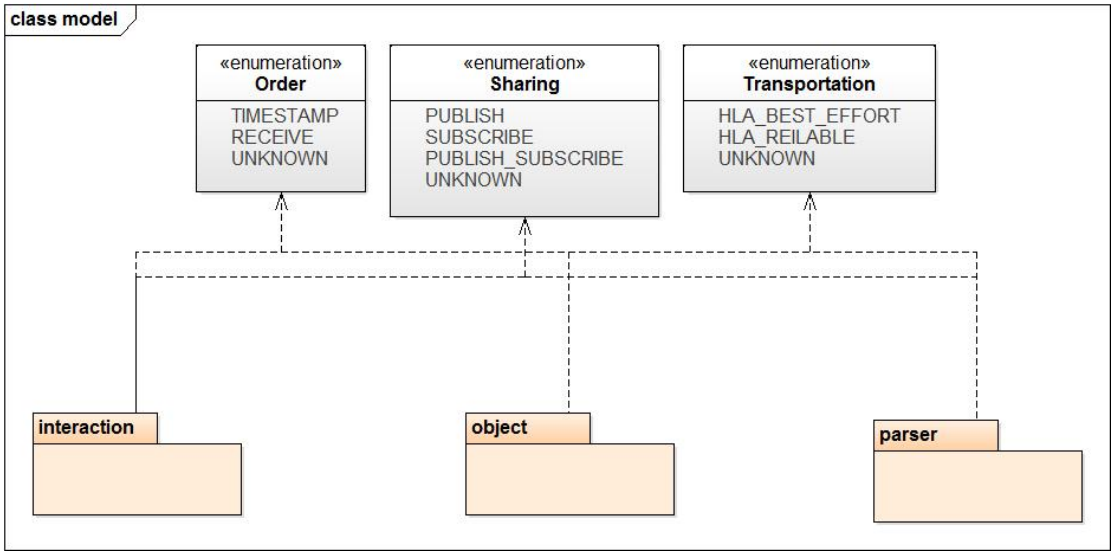


**Figure 11: The UML class diagram of the dkf.model package.**

This package contains classes to manage an *ObjectModel (ObjectClass* and *InteractionClass)*. In particular, the DKF framework uses these classes to facilitate the subscribe, publication and the data update of an *ObjectModel* provided by the user.

This package also includes a set of sub-packages each implementing specific services.

#### 3.1.6.1 The dkf.model.parser package

The dkf.model.parser defines the parser for the *ObjectModel (ObjectClass* and *InteractionClass)*. The parser is used by the DKF to determine the structure of the *ObjectModel,* in order to retrieve its structure and the values of its fields before publishing/updating on HLA/RTI platform. The architecture of the sub-package is shown in Figure 12 by using a UML Class Diagram.

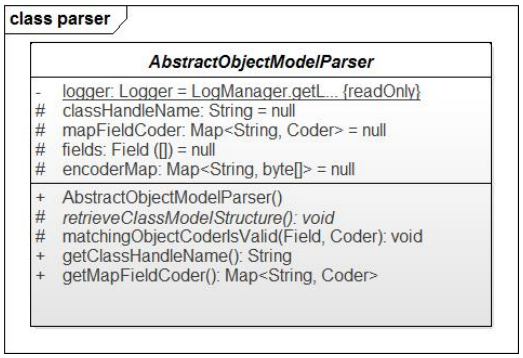


**Figure 12: The UML class diagram of the dkf.model.parser package.**

#### 3.1.6.2 The dkf.model.object package

This package provides some classes to be used to define an *ObjectClass*; it uses the dkf.model.object.annotation package (see Figure 14), which defines two *Annotation* interfaces, for creating an *ObjectClass*. In particular, these interfaces are used by the programmer to attach metadata to an *ObjectClass* class.

The architecture of the dkf.model.object package is shown in Figure 13 by using a UML Class Diagram.

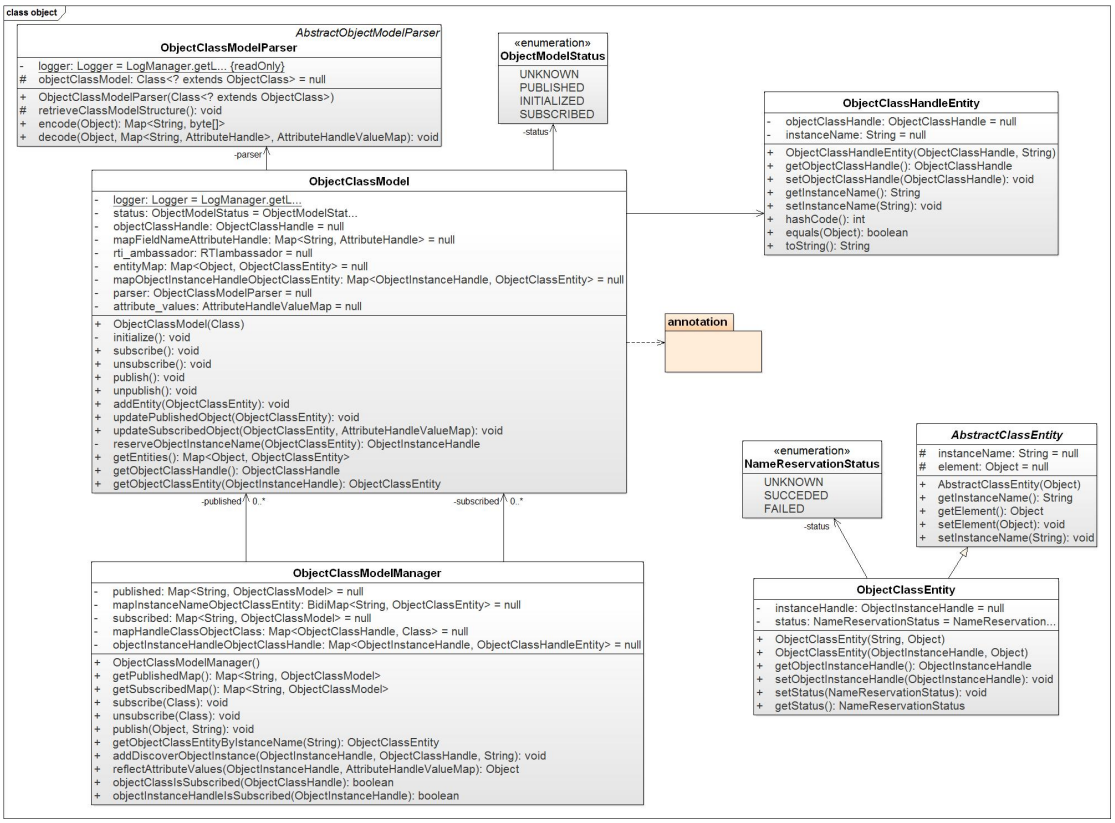


**Figure 13: The UML class diagram of the dkf.model.object package.**

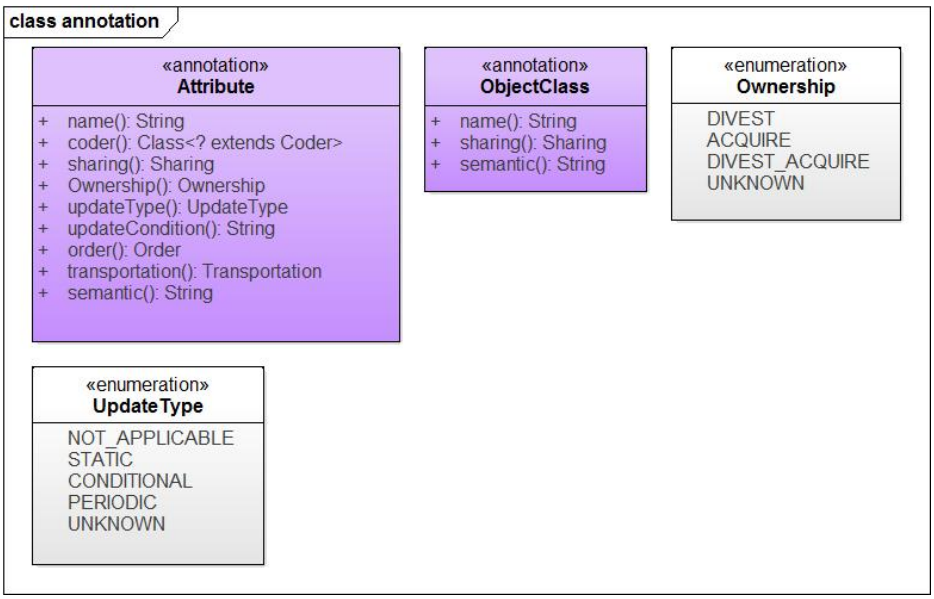


**Figure 14: The UML class diagram of the dkf.model.object.annotation package.**

### 3.1.6.3 The dkf.model.interaction package

It provides some classes used by the user to define an *InteractionClass*. It uses the dkf.model.interaction.annotation package (see Figure 16), which defines two *Annotation* interfaces that are used by the programmer to attach metadata to an InterationClass class.

The architecture of the dkf.model.interaction package is shown in Figure 15 by using a UML Class Diagram.

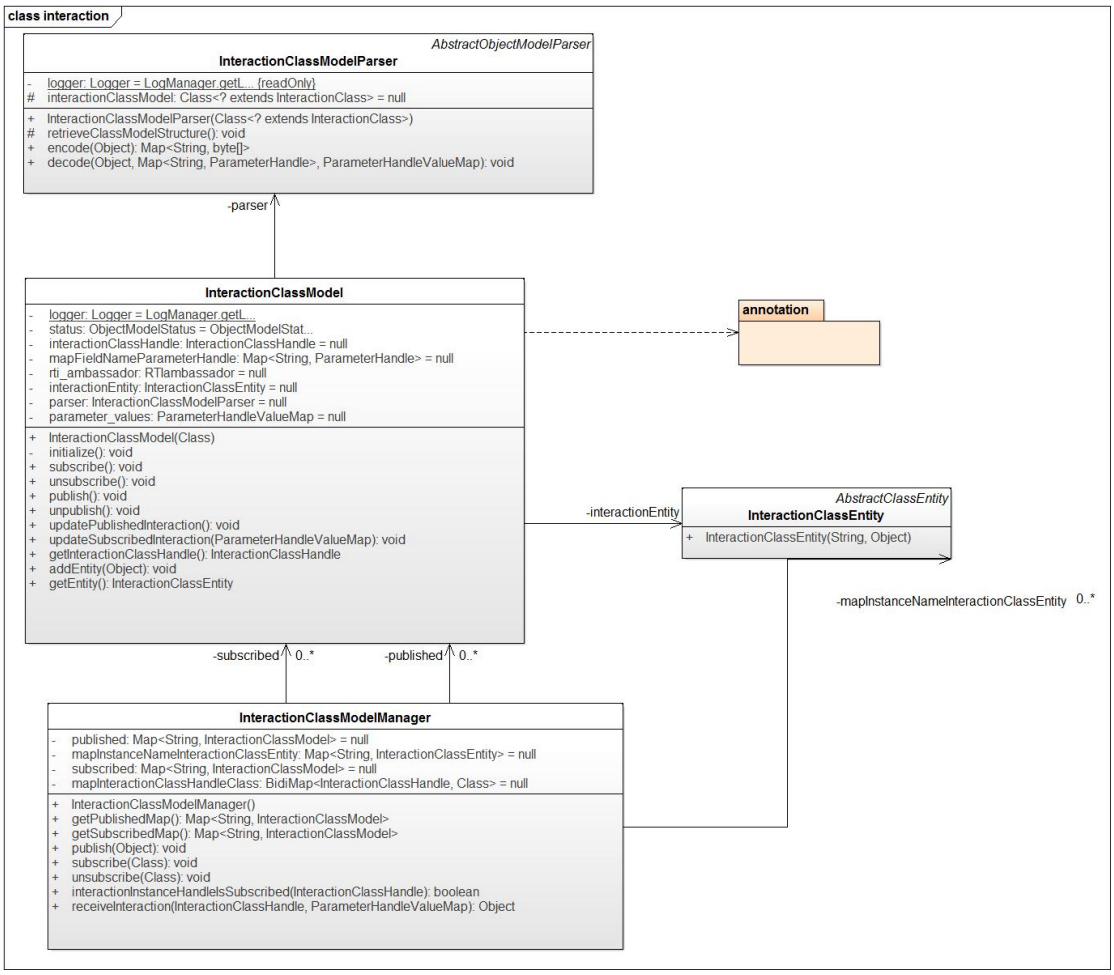


**Figure 15: The UML class diagram of the dkf.model.interaction package.**

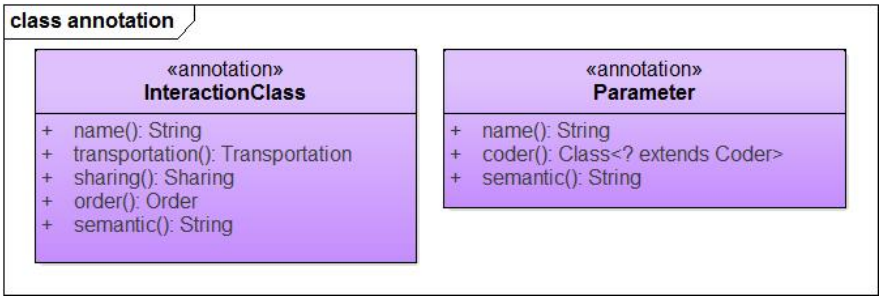


**Figure 16: The UML class diagram of the dkf.model.interaction.annotation package.**

### 3.1.7 The dkf.time package

It provides some classes that are used by the classes in the *core* package to properly instantiate the simulation time, according to a FOM. The architecture of the dkf.time package is shown in by using a UML Class Diagram.

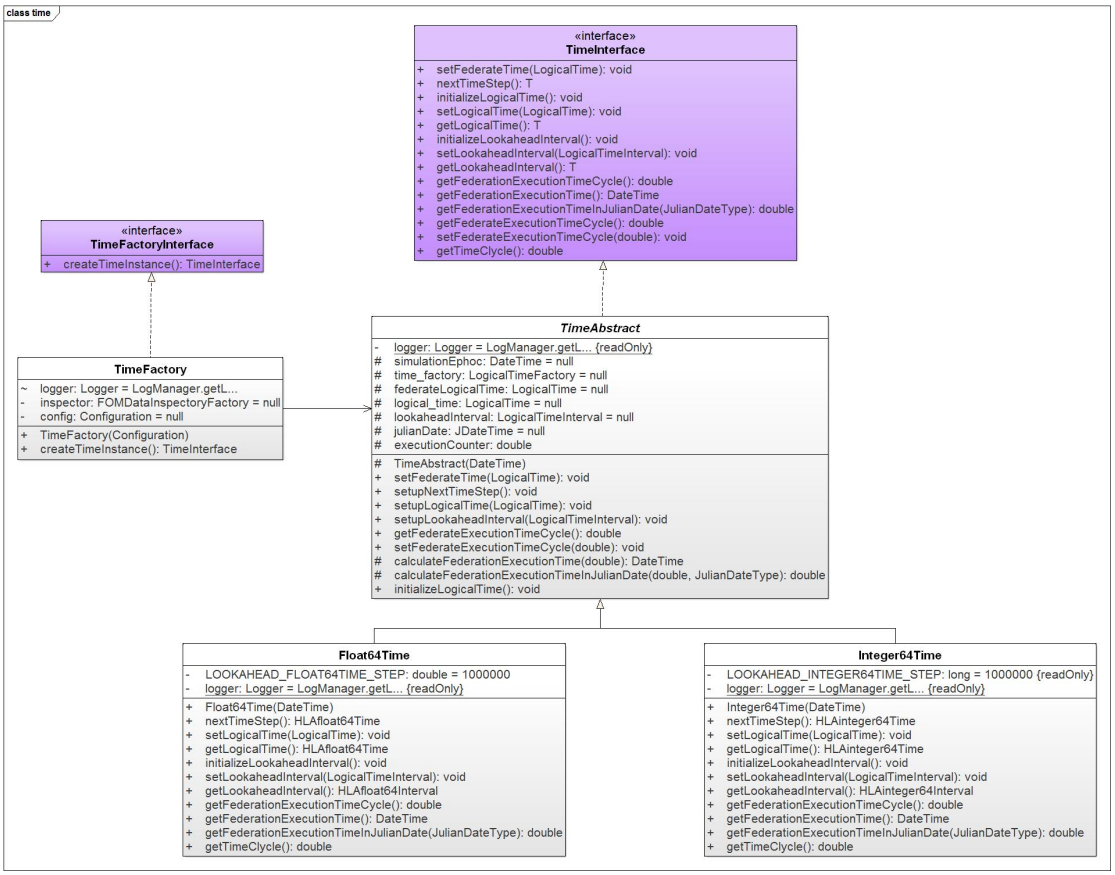


**Figure 17: The UML class diagram of the dkf.time package.**

# 4 The Behavioral Model of a DKF-based Federate

The *DKFAbstractFederate* class also provides and manages the life cycle of a Federate according to the behavioral model that is shown in Figure 18 through a UML Statechart diagram. As a consequence, a working team has only to define the specific behavior of its Federate without worrying about low-level implementation details since the DKF manages them.

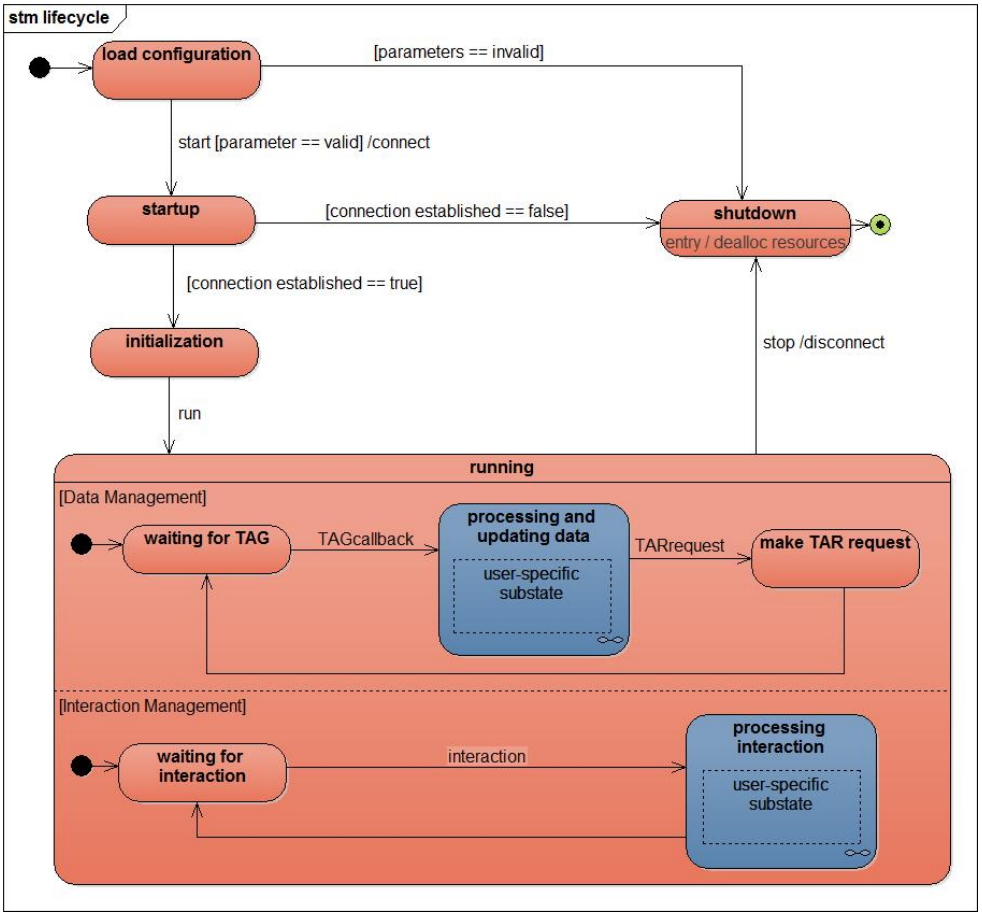


**Figure 18: The lifecycle of a DKF-based Federate.**

In the "*load configuration*" state, the DKF loads the configuration parameters from a .json file. A transition to the "*startup*" state happens if the configuration parameters are valid and during the state transition a connection to the Federation

execution is performed. Otherwise, if the configuration parameters are invalid a state transition to the "*shutdown*" state is performed. In this latter state, all the resources engaged by the DKF classes are de-allocated and the lifecycle terminates. In the "*startup*" state, the connection status is checked. If the connection is not established the lifecycle ends with a transition to the "*shutdown*" state.

Otherwise, a transition to the "*initialization*" state is performed; in this state, the Federate could perform additional operation for exchanging initialization objects (e.g. publishing and subscribing information) before entering the running state (and thus the time advancement loop: "*waiting for a Time Advance Grant (TAG)*" → "*processing and updating data*" → "*make a Time Advance Request (TAR*)"). After that, the time management thread is activated and a transition to the "*running*" state is performed. The running state is composed by two sub-states operating in an AND-decomposition fashion. The *"Data Management"* sub-state deals with the pro-active part of the Federate behavior through three states: (i) "*waiting for TAG*": the DKF waits for the "TAG (Time Advance Grant) Callback" from the RTI. When the callback is received a transition to the *"processing and updating data"* state is performed; (ii) *"processing and updating data"*: the "logical time" is updated, the pro-active behavior of the specific Federate defined by the working team is executed, and then a transition to the *"make TAR request"* state is performed; (iii) *"make TAR request"*: the DKF requests to the RTI the grant for the next "logical time".

The *"Interaction Management"* sub-state deals with the re-active part of the behavior of the Federate: upon reception of RTI callback related to subscribed elements, a transition to the "*processing interaction*" state is performed where the received information is handled.

When the simulation ends a transition from the "*running*" state to the "*shutdown*" state is performed and, during the state transition, the HLA Federate is disconnected from the RTI.

## 4.1 Concurrency aspects

The Federate life cycle provided by the current version of the DKF, v 0.0.1 (see Figure 18) brings many advantages to programmers for easily getting a ready-to-run HLA Federate. In particular, a programmer has only to specify the specific behavior of its Federate by defining: (i) the *proactive behavior* of the Federate, through the *"processing and updating data"* composite state, which is accessed between a TAG (Time Advance Grant) and a TAR (Time Advance Request); and (ii) the *reactive behavior* of the Federate, by specifying the *"processing interaction"* composite state so as to indicate how to handle the RTI callbacks about interactions/objects that the Federate has subscribed.

However, it is worth noting that these two composite states (*"processing and updating data"* and *"processing interaction"*) are in *AND decomposition* and thus concurrently executed. As a consequence, the concurrency aspects that could rise have to be properly managed by the programmer. The DKF will provide specific services and support to handle these concurrency aspects in its future releases.

# 5 Exploiting the HLA Development Kit

This section presents a case study concerning the development of a HLA Federate by using the *HLA Development Kit* in the context of the Simulation Exploration Experience (SEE) 2015 project [7]. In particular, the architecture and behavior of the developed and experimented HLA Federate are based on a specific Space domain-extension of the *HLA Development Kit*, called the *SEE HLA Starter Kit* (available on Google Code [6], see[1],[1] for details).

## 5.1 The SEE HLA Starter Kit

To promote the adoption and experimentation of the *HLA Development Kit* and its *DKF*, the Kit has been specialized in the *SEE HLA Starter Kit (SKF)* with the aim to ease the development of HLA Federates in the context of the Simulation Exploration Experience (SEE) project [7]. SEE is an event organized by the Simulation Interoperability Standards Organization (SISO), in collaboration with NASA and other research and industrial partners, with the objective to promote the adoption of the HLA standard and compliant tools by involving university teams in the distributed simulation. The SEE-specific features introduced in the SKF (as an example the implementation of SEE Dummy and Tester Federates and the publishing/subscribing of *ReferenceFrames*) aim not only at reducing the development efforts but also at improving the reliability of SEE Federates and thus reducing the problems arising during the final integration and testing phases of the SEE project [7]. Moreover, this Space domain-extension of the DKF allows to prove how, starting from a domain-independent core of the DKF, conceived for supporting the development of general-purpose HLA Federate, it is possible to easily add and integrate application-specific extensions for supporting the development of domain-specific Federates.

The following subsections are devoted to present the process to build a Federate. with reference to its SEE-specific extension (the *SEE-SKF: SEE HLA Starter Kit Framework*). Figure 19 shows the architecture of a *DKF/SEE-SKF* based Federation.

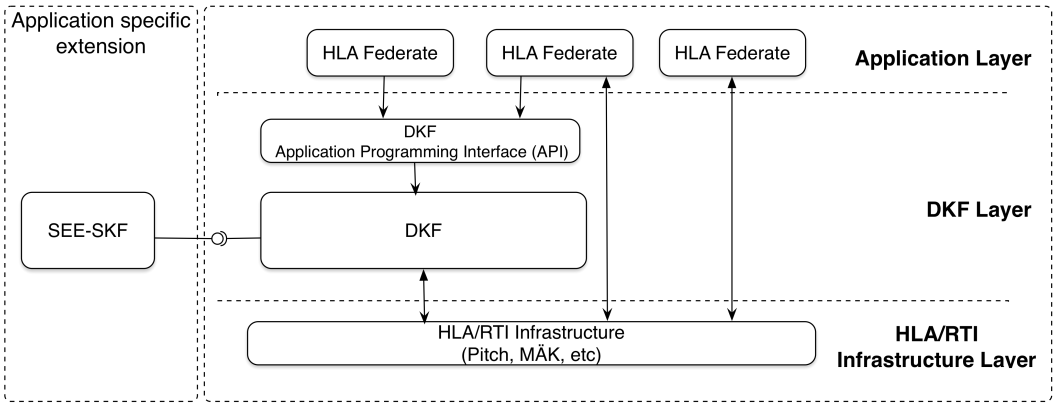


**Figure 19: The architecture of a DKF/SEE-SKF based Federation.**

In order to clearly explain all the steps of the development process, for each step the Java code is reported.

## 5.2 The development process of a DKF/SEE-SKF based Federate

The process to build a Federate from scratch by using the *DKF/SEE-SKF* is composed by the following four main steps:

1. Build a model of the Federate that specifies: the objects that the Federate manages (as specified in the FOM), the attributes of these objects and the coders to handle such attributes. It is possible to use the set of basic coder provided by the *SEE-SKF* or simply implement new coders by using the *SEE-SKF* classes; other available coders can be also exploited;
2. Build a concrete Federate that specifies the behavior of the model defined at step (i). It is required to extend the *SEEAbstractFederate* abstract class provided by the *SEE-SKF* and implement three methods according to the

Federate life-cycle that is provided and completely managed by the *DKF*/*SEE-SKF* (see Fig. 4), specifically: (a) a method for initialization operations before entering in the *"running state"* (a *configureAndStart()* method); (b) a method for specifying the pro-active part of the behavior of the Federate (*doAction()* method) and that is executed between a TAG and a TAR; (c) a method (*update(…)* method) that specifies the re-active part of the behavior of the Federate, i.e. how to handle the RTI callbacks about the interactions/objects that the Federate has subscribed;

3. Implement the Federate Ambassador. This step requires extending the *SEEAbstractFederateAmbassador*; typically, since no specific implementation is required, the child class has only to define its constructor which in turn calls the parent one: all the typical Ambassador's features are provided and managed by the *DKF*/*SEE-SKF*;
4. Implement a main class so as to instantiate and run the developed Federate.

## 5.3 Using the DKF/SEE-SKF framework: the "TestFederate"

The architecture of the *TestFederate* is shown in Figure 20 by using a UML Class Diagram. The Java code of the classes related to the *TestFederate* is reported in the following.

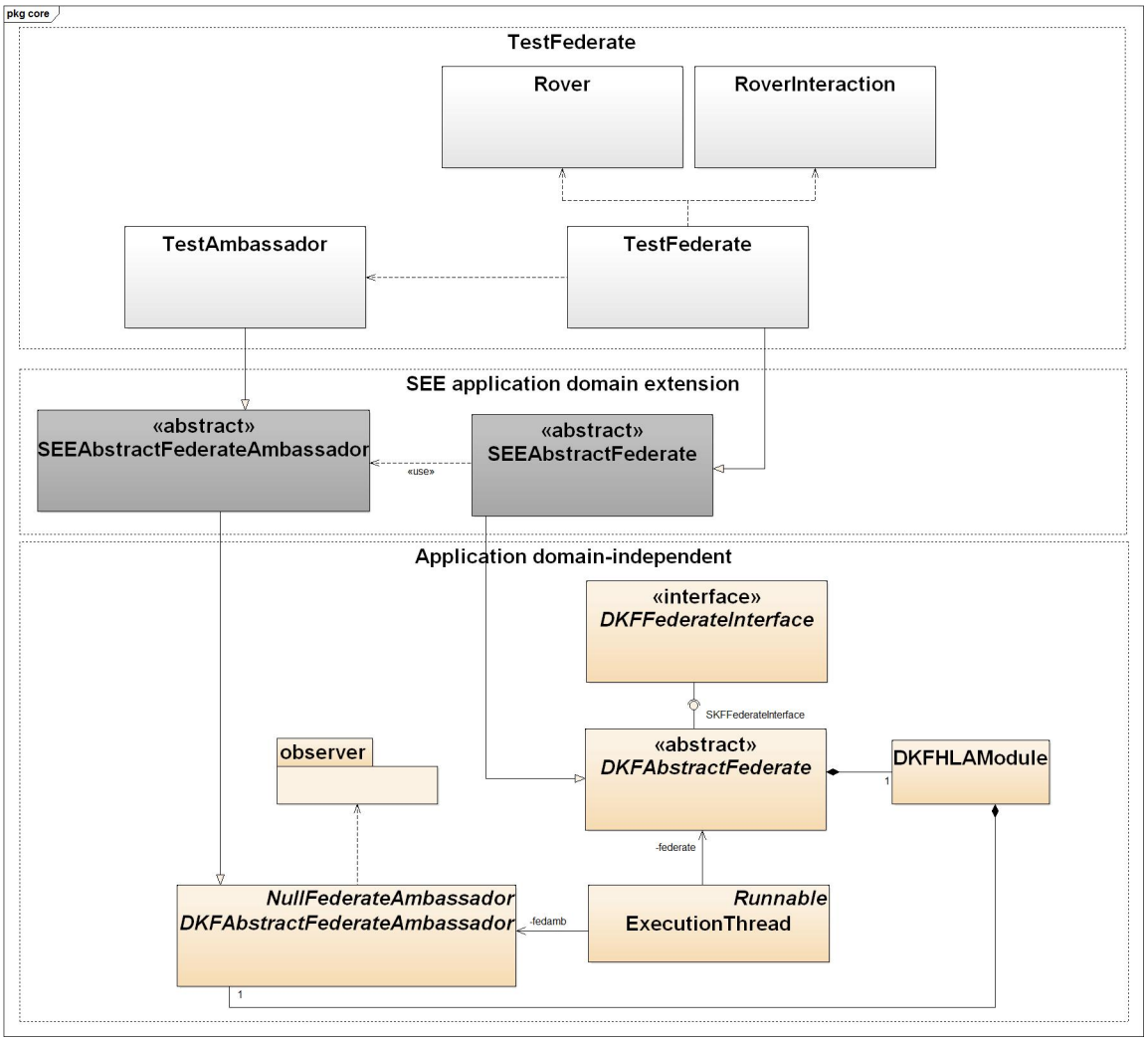


**Figure 20: The UML class diagram of the "TestFederate".**

### 5.3.1 The Rover class (ObjectClass)

```
(see Note [A])
package model;

import skf.coder.HLAinteger32BECoder;
import skf.coder.HLAunicodeStringCoder;
import skf.model .annotations.Attribute;
import skf.model .annotations.ObjectClass;

@ObjectClass(name = "PhysicalEntity.Rover") (see Note [B])
public class Rover {
	private static int roverIDCounter = 0;

	@Attribute(name = "id", coder = HLAinteger32BECoder.class)
	private Integer identifier;

	@Attribute(name = "name", coder = HLAunicodeStringCoder.class)
	private String name = null;

	public Rover() {}

	public Rover(String name) {
		this.identifier = roverIDCounter;
		roverIDCounter++;
		this.name = name;
	}

	public Integer getIdentifier() {
		return identifier;
	}

	public void setIdentifier(Integer identifier) {
		this.identifier = identifier;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Override
	public String toString() {
		return "Rover [identifier=" + identifier + ", name=" + name + "]";
	}
```

```
}
```

### 5.3.2 The Interaction class (InteractionClass)

```
(see Note [A])
package model;

import skf.coder.HLAunicodeStringCoder;
import skf.model.interaction.annotations.InteractionClass;
import skf.model.interaction.annotations.Parameter;

@InteractionClass(name = "RoverInteraction") (see Note [C])

public class Interaction {

	@Parameter(name = "name", coder = HLAunicodeStringCoder.class)
	private String name = null;

	@Parameter(name = "payload", coder = HLAunicodeStringCoder.class)
	private String payload = null;

	public Interaction() {}

	public Interaction(String name, String payload) {
		this.name = name;
		this.payload = payload;
	}

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}

	public String getPayload() {
		return payload;
	}

	public void setPayload(String payload) {
		this.payload = payload;
	}

	@Override
	public String toString() {
		return "Interaction [name=" + name + ", payload=" + payload + "]";
```

```
        }
}
```

### 5.3.3 The TestFederate class

```
package federate;

import hla.rti1516e.exceptions.*;
import java.net.MalformedURLException;
import java.util.LinkedList;
import java.util.List;
import java.util.Observable;
import java.util.Observer;
import java.util.Scanner;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import model.Rover;
import model.Interaction;
import siso.smackdown.FrameType;
import siso.smackdown.ReferenceFrame;
import skf.config.Configuration;
import skf.core.SEEAbstractFederate;
import skf.core.SEEAbstractFederateAmbassador;
import skf.exception.UnsubscribeException;
import skf.exception.UpdateException;
import skf.utility.JulianDateType;
import skf.utility.TimeUnit;
import skf.utility.TimeUtility;


public class TestFederate extends SEEAbstractFederate implements Observer {

        private Logger logger = LogManager.getLogger(TestFederate.class);

        /*
         * For MAK local_settings_designator = "";
         * For PITCH local_settings_designator = "crcHost=" + <crc_host> + "\ncrcPort=" + <crc_port>;
         */
        private static final String local_settings_designator = "";

        private List<Rover> listRover = new LinkedList<Rover>();
        private ReferenceFrame rf = null;
        private Interaction interaction = null;
```

```
	public TestFederate(SEEAbstractFederateAmbassador seefedamb) {
		super(seefedamb);
	}

	public void configureAndStart(Configuration config) throws Exception {

		// 1. configure the DKF framework
		super.configure(config);
		// 2. Connect on RTI
		super.connectOnRTI(local_settings_designator);
		// 3. The Federate joins into the Federation execution
		super.joinIntoFederationExecution();

		try {
			// 4. Publish/Subscribe
			super.subscribeSubject(this);
			// 5. Publish/Subscribe objects and/or interaction
			super.subscribeReferenceFrame(FrameType.MarsCentricInertial);
			super.subscribeReferenceFrame(FrameType.EarthCentricFixed);
			listRover.add(new Rover("lunarRover1","lunarRover1"));
			listRover.add(new Rover("lunarRover2","lunarRover2"));

			for(Rover r : listRover)
				super.publishElement(r, r. getName());

			interaction = new Interaction("interaction", "payload");
			super.publishInteraction(interaction);

		} catch (Exception e) {
			e.printStackTrace();
		}

		// 6. Execution-loop
		super.startExecution();

		try {
			logger.info("Press any key to disconnect the federate from the federation
execution");
			new Scanner(System.in).next();
			stop();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	@Override
```

```
	public void update(Observable o, Object arg) {
		this.rf = (ReferenceFrame) arg;
		logger.info(rf);

	}

	private int count = 1;

	@Override
	protected void doAction() {
		logger.info("federation execution time cycle:
"+TimeUtility.convert(super.getTime().getFederationExecutionTimeCycle(), TimeUnit.MICROSECONDS,
TimeUnit.SECONDS));
		logger.info("federate time cycle:
"+TimeUtility.convert(super.getTime().getFederateExecutionTimeCycle(), TimeUnit.MICROSECONDS,
TimeUnit.SECONDS));

		logger.info("federate time date: "+super.getTime().getFederationExecutionTime());

		logger.info("federation execution jd:
"+super.getTime().getFederationExecutionTimeInJulianDate(JulianDateType.DATE));
		logger.info("federation execution mjd:
"+super.getTime().getFederationExecutionTimeInJulianDate(JulianDateType.MODIFIED));
		logger.info("federation execution rjd:
"+super.getTime().getFederationExecutionTimeInJulianDate(JulianDateType.REDUCED));
		logger.info("federation execution tjd:
"+super.getTime().getFederationExecutionTimeInJulianDate(JulianDateType.TRUNCATED));

		try {

			Rover tmp = null;

			for(int i=0; i<listRover.size(); i++){
				tmp = listRover.get(i);
				// update the Identifier of the Rover
				tmp.setIdentifier(count);
				// update on RTI the rover
				super.updateElement(tmp);
				count++;
			}

			//update the 'payload' of the RoverInteraction
			interaction.setPayload("payload"+Math.random());
			super.updateInteraction(interaction);


		} catch (Exception e) {
```

```
                    e.printStackTrace();
                }

        }

        public void stop() throws Exception {

                // Unsubcribe from the Subject
                super.unsubscribeSubject(this);

                // Disconnect from the HLA/RTI platform
                super.diconnectFromRTI();
        }

}
```

### 5.3.4 The TestAmbassador class

```
import skf.core.DKFAbstractFederateAmbassador;

public class TestAmbassador extends SKFAbstractFederateAmbassador {
        public TestAmbassador() {
                super();
        }
}
```

### 5.3.5 The Main class

```
import java.io.File;
import java.io.FileNotFoundException;

import federate.TestAmbassador;
import federate.TestFederate;
import skf.config.Configuration;
import skf.config.ConfigurationFactory;

public class MainTest {

        private static final File configurationFile = new File("testResources/configuration/conf.json");

        public static void main(String[] args) throws FileNotFoundException {

                TestAmbassador testfedamb = new TestAmbassador();
                TestFederate testfed = new TestFederate(testfedamb);
                ConfigurationFactory factory = null;
                Configuration config = null;
```

```
                try {
                        factory = new ConfigurationFactory();
                        config = factory.importConfiguration(configurationFile);
                        testfed.configureAndStart(config);

                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

NOTE:

[A]. In order to be processed by the DKF, the ObjectClass/InteractionClass class must be a JavaBean. It is a specially constructed Java class coded according to the JavaBeans API specifications (for more details see the website: http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/). The main characteristics that distinguish a JavaBean from other Java classes are the following:

- A JavaBean provides a default, no-argument constructor;
- A JavaBean should be serializable and implement the Serializable interface. (this characteristic is no necessary for the Kit);
- A JavaBean may have a number of properties which can be read or written;
- A JavaBean may have a number of "getter" and "setter" methods for the properties.

[B]. The value of the attribute 'name' must match with the path defined in its FOM file, starting from the tag <name>HLAobjectRoot</name> and with the root not included (in this case, PhysicalEntity.Rover).

[C]. The value of the attribute 'name' must match with the path defined in its FOM file, starting from the tag <name>HLAinteractionRoot</name> and with the root not included (in this case, RoverInteraction).

### 5.3.6 The Rover FOM

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<objectModel
        xsi:schemaLocation="http://standards.ieee.org/IEEE1516-2010
http://standards.ieee.org/downloads/1516/1516.2-2010/IEEE1516-DIF-2010.xsd"
        xmlns="http://standards.ieee.org/IEEE1516-2010"   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
        <modelIdentification>
                <name>MyModule</name>
                <type>FOM</type>
                <version>1.0</version>
                <securityClassification>unclassified</securityClassification>
                <purpose></purpose>
                <applicationDomain></applicationDomain>
                <description>Description of MyModule</description>
                <useLimitation></useLimitation>
                <other></other>
        </modelIdentification>
        <objects>
                <objectClass>
                        <name>HLAobjectRoot</name>
                        <objectClass>
                                <name>PhysicalEntity</name>
                                <objectClass>
                                        <name>Rover</name>
                                        <sharing>PublishSubscribe</sharing>
                                        <semantics></semantics>
                                        <attribute>
                                                <name>id</name>
                                                <dataType>HLAinteger16BE</dataType>
                                                <updateType>Static</updateType>
                                                <updateCondition>NA</updateCondition>
                                                <ownership>NoTransfer</ownership>
                                                <sharing>PublishSubscribe</sharing>
                                                <transportation>HLAreliable</transportation>
                                                <order>Receive</order>
                                                <semantics>The identifier of the ROVER
                                                module.</semantics>
                                        </attribute>
                                        <attribute>
                                                <name>name</name>
                                                <dataType>HLAunicodeString</dataType>
                                                <updateType>Static</updateType>
                                                <updateCondition>NA</updateCondition>
                                                <ownership>NoTransfer</ownership>
                                                <sharing>PublishSubscribe</sharing>
                                                <transportation>HLAreliable</transportation>
```

```
                                    <order>Receive</order>
                                    <semantics>The name of the ROVER module.</semantics>
                                </attribute>
                            </objectClass>
                        </objectClass>
                    </objectClass>
                </objects>
                <interactions>
                    <interactionClass>
                        <name>HLAinteractionRoot</name>
                        <interactionClass>
                            <name>RoverInteraction</name>
                            <sharing>PublishSubscribe</sharing>
                            <transportation>HLAreliable</transportation>
                            <order>Receive</order>
                            <semantics></semantics>
                            <parameter>
                                <name>name</name>
                                <dataType>HLAunicodeString</dataType>
                                <semantics></semantics>
                            </parameter>
                            <parameter>
                                <name>payload</name>
                                <dataType>HLAunicodeString</dataType>
                                <semantics></semantics>
                            </parameter>
                        </interactionClass>
                    </interactionClass>
                </interactions>
                <dataTypes>
                    <simpleDataTypes />
                    <enumeratedDataTypes />
                    <arrayDataTypes />
                    <fixedRecordDataTypes />
                    <variantRecordDataTypes />
                </dataTypes>
                <notes />
            </objectModel>
```

### 5.3.7 The Configuration file

```
{
  "asynchronousDelivery" : true,
  "crcHost" : "localhost",
  "crcPort" : 8989,
  "federateName" : "TestFederate",
  "federateType" : "Test",
  "federationName" : "Test Federation",
```

```
  "fomDirectory" : "C: \\foms",
  "realtime" : false,
  "simulationEphoc" : "2015-04-12T20:00:00.000Z", (see Note [D])
  "timeConstrained" : true,
  "timeRegulating" : false
}
```

NOTE:

[D]. The Simulation Ephoc is expressed according to the ISO 8601 standard, for more detail see the website:

http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=40874.

## Appendix A: The main services of the HLA Development Kit

For the definition of the "HLA Development Kit" and related development framework (DKF), some important "services" to be provided are under identification. In the following, a starting list is reported by organizing them into categories and by also specifying the related RTI service type as defined by the HLA standard.

**CONNECTION MANAGEMENT services *(RTI-Coordination Services)*:**

- management of the connection parameters;
- management of the connection of the federates to a Federation execution;
- management of the resign of the federates from a Federation execution.

**FOM MANAGEMENT services *(RTI-Information Services)*:**

- FOM checking;
- FOM module publication services.

**SIMULATION TIME MANAGEMENT services *(RTI-Synchronization Services)*:**

- simulation time handling;
- time standard conversions.

**DATA DISTRIBUTION MANAGEMENT services *(RTI-Information Services)*:**

- Discovering Object;
- Subscribing Object, Interaction, Classes;
- Updating and reflecting Object attibutes;
- Sending and receiving Interactions;
- Removing discovered Object.

**LOGGING services:**

- management of DKF-specific logs.

**TESTING services:**

- IP Configuration testing;
- MS Windows Firewall testing;
- LRC/CRC parameters testing.

## References

[1] Anagnostou, A., Chaudhry, N.R., Falcone, A., Garro, A., Salah, O., Taylor, S.J.E., *Easing the development of HLA Federates: the HLA Development Kit and its exploitation in the SEE Project*. In Proc. of the 19th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (ACM/IEEE DS-RT), Chengdu, China, October, 14-16, IEEE Computer Society, (2015).

[2] Anagnostou, A., Chaudhry, N.R., Falcone, A., Garro, A., Salah, O., Taylor, S.J.E., *A Prototype HLA Development Kit: Results from the 2015 Simulation Exploration Experience*. In Proc. of the ACM SIGSIM PADS 2015, London, UK, June, 10-12, (2015).

[3] Bocciarelli, P., D'Ambrogio, A., Falcone, A., Garro, A., Giglio, A., *A model-driven approach to enable the distributed simulation of complex systems.* In Proc. of the 6th Complex Systems Design & Management (CSD&M) 2015, Paris, France, November 23-25, (2015).

[4] Falcone, A., Garro, A., Longo, F., Spadafora, F., *SimulationExploration Experience: A Communication System and a 3D Real Time Visualization for a Moon base simulated scenario*. In Proc. of the 18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (ACM/IEEE DS-RT), Toulouse, France, October, 1-3, IEEE Computer Society, (2014).

[5] HLA Development Kit, [online], available https://smash-lab.github.io/HLA-Development-Kit/.

[6] SEE HLA Development Kit, [online], available https://code.google.com/p/see-hla-starterkit/.

[7] Simulation Exploration Experience (SEE) project, [online], available http://www.exploresim.com/.