5.0 Declaration Management

This chapter introduces the RTIambassador service and FederateAmbassador callback methods that support declaration management. Declaration management includes publication, subscription and supporting control functions. Federates that produce objects (or object attributes) or that produce interactions must declare exactly what they are able to publish (i.e., generate). Federates that consume objects (or object attributes) or that consume interactions must declare their subscription interests. (See Figure 5.1.)

---

Declaration Management

- Declaration Management serves the purpose of coordinating data exchange between federates.

- Declaration Management:
  - Specifies the data a federate will send and receive.
  - Controls where data is sent based on external interest.
The RTI keeps track of what participating federates can produce and what they are interested in consuming and sends control signals to throttle what is produced based on consumer interest. As depicted in Figure 5.2, the RTI uses control signals to inform producers exactly what they should transmit. The goal is to keep traffic off the communications network and off nodes supporting federates that don’t need the data.

5.1 Object Vocabulary Review

It's worth a moment to review some basic HLA terminology. (See Figure 5.3.) Object classes are comprised of attributes. Object classes describe types of things that can persist. For example, "traffic light" might be an object class. Objects of type traffic light have certain attributes (e.g., color, size, and duration). Actual, simulated traffic lights are instances of the object class traffic light. The term "object" standing alone is sometimes used to describe an instance of a particular object class, but sometimes refers to the type information. Object classes may be related to cookie cutters and object instances to the cookies produced using the cookie cutters.
Object Vocabulary

- Object Classes
  - Comprised of Attributes
  - Describe Types of Things That Can **Persist**
- Interaction Classes
  - Comprised of Parameters
  - Describe Types of Events (Transitory)

- **Objects Persist, Interactions Do Not**

*Interaction classes* are comprised of parameters. Interaction classes describe *types* of events. Interaction instances are specific events. It's fair to say, "Objects are similar to interactions in so much as objects are comprised of attributes and interactions are comprised of parameters." The HLA recognizes this inherent symmetry and leverages it when appropriate. The primary difference between objects and interactions is **persistence**. Objects persist, interactions do not.

Would an airliner be described by an object class or an interaction class? The answer depends on the purpose of the simulation. A simulation that focuses on tourism may have no interest the flight characteristics of the airliner itself, but only in the number of tourists that are arrive on a given flight. The dynamics of the flight may be of no importance whatever to the simulation. Here, the flight could be modeled as an interaction – possibly between the countries of departure and arrival. Another simulation may focus on the in-flight characteristics of the airliner. The fact that the airliner carries tourists may be incidental. Here, the airliner persists and should be modeled as an object.
5.2 Object Hierarchies

Figure 5.4 illustrates a class hierarchy and accompanying Venn diagram. Object classes and interaction classes can be constructed hierarchically. For example, assume that objects of type $W$ are comprised of the attributes "a," "b," "c," and "d" – abbreviated "{a, b, c, d}." It is possible to define object classes that extend object class $W$. Object class $W$ is extended to produce the object classes $X$ and $Z$. Object class $X$ is further extended to produce the object class $Y$.

![Class Hierarchy - Venn Diagram](image)

Object-oriented programming enthusiasts will recognize such hierarchical representations. Various communities use different phrases to describe object hierarchies. Some examples include:

- $X$ extends $W$.
- $W$ is a base type.
- $X$ is derived from $W$.
- $Y$ inherits from $X$.
- $W$ is an ancestor of $Z$.
- $X$ is a child of $W$.

---

1 Developers with a strong object-orientation should note that HLA "objects" are defined primarily by their constituent data elements rather than on behavior. In this way, HLA "objects" have more in common with relational models than object-oriented models.
Y is a descendant of W. Y and Z are leaf objects.

W is the parent of Z.

The basic idea is that when an object class is extended to produce a new object class, the new object class contains all the attributes of the class being extended and possibly more. The object diagram and Venn diagram (Figure 5.4) illustrate the relationship between the object classes W, X, Y, and Z. Object class W has the four attributes \{a, b, c, d\}, class X adds the attributes \{e, f, g\} so instances of class X have attributes \{a, b, c, d, e, f, g\}.

5.3 Publishing and Subscribing Objects

Each federate must publish the object classes and interaction classes it plans to produce. It is possible for a federate to publish a subset of the available attributes for a given object class.

5.3.1 Object Publication

The object class Y contains the attributes \{a, b, c, d, e, f, g, h\}. A federate can create instances of object class Y, without specifying all of the attributes associated with Y. For example, Y might be a particular kind of aircraft. A given federate may know some information about aircraft instances (e.g., position information), but rely on other federates to "fill in" the missing pieces (e.g., intelligence about the aircraft). In such a case, the federate would indicate that it can publish particular attributes associated with Y. In Figure 5.5, Federate #1 indicates that it can publish attributes \{b, e, f, g, h\} for object class Y.

Each federate must indicate explicitly which attributes it can produce (i.e., introduce or update) on a per class basis. Multiple federates may be able to publish Y instances. Federate #3 might publish all of the attributes associated with object class Y. Federate #7 may be able to publish attributes \{a, c, f\} for Y.
An implicit attribute, known by the name "privilegeToDelete," is included whenever a publication capability is registered for an object class. Only the federate that created a particular object instance is allowed to delete the instance unless the privilege to delete is conveyed to another Federate. Ownership Management, services provide the ability to exchange attribute update and object deletion responsibility among federates.

A federate must explicitly state every object class it intends to produce via the RTIambassador’s publishObjectClass() method. A separate call to publishObjectClass() is required for every object class including objects that appear in class hierarchies. If Federate #9 wishes to produce instances of object classes W, X, Y, and Z, it must say so explicitly using four publication calls.

5.3.2 Interaction Publication

As with object classes, each federate must state explicitly which interaction classes it intends to produce using the publishInteractionClass() method (Figure 5.6). Interactions are produced as "all or nothing." It isn’t possible to specify which parameters in an interaction will be published. If a federate indicates that it intends to publish an
interaction, it should be capable of specifying all parameters associated with the interaction.

5.3.3 Object Subscription

Federates indicate their interest in certain object classes via the `RTIambassador` method `subscribeObjectClassAttributes()`. Object subscription differs from object publication. When a federate subscribes to an object class, it is expressing an interest in learning about all object instances of the class. For example, a federate subscribing to object class X (as shown in Figure 5.4) will discover all instances of class X produced by other federates in the federation. Additionally, a federate subscribing to X will discover all instances of class Y (produced by other federates) as though they were instances of class X. This is an example of type promotion, shown in Figure 5.7.
Whenever a federate expresses a subscription interest in a particular object class, the RTI presumes that the federate is interested in instances of the descendant classes as well. A federate subscribing to class $W$ would see external instances of classes $X$, $Y$, and $Z$ as instances of class $W$. This can be a useful tool. Class $W$ might represent all aircraft. Class $X$ might represent commercial aircraft, while class $Y$ represents small private aircraft. A federate may wish to know about all aircraft, but not care about the details – including the private vs. commercial designation.

A federate is informed about a new object instance if (a) the federate has subscribed to the object class of the instance or (b) the instance can be promoted (i.e., up the hierarchy) to a subscribed object class. When an object is promoted, attributes particular to the original class are dropped. An instance of object class $Y$ has attributes $\{a, b, c, d, e, f, g, h\}$. A federate subscribing to object class $X$ can discover the $Y$ instance as an $X$. Since attribute "h" is not present in instances of class $X$, that information is lost.

\[\text{Some attributes may not have assigned values. It depends on what the originating federate has published for this object and the extent to which other federates have contributed to what's known about the instance.}\]
A federate can subscribe to multiple classes in a class hierarchy. If a federate subscribed to class \( W \) and \( X \), the following would be true: (Figure 5.8)

1. Instances of object class \( W \) would be seen without promotion.
2. Instances of object class \( X \) would be seen without promotion.
3. Instances of object class \( Y \) would be seen as instances of object class \( X \).
4. Instances of object class \( Z \) would be seen as instances of object class \( W \).

**Interaction Subscription**

- Each federate subscribes to the interaction classes it wishes to receive.
- It is not possible to subscribe to individual parameters of an interaction class.
- A federate is informed about a new interaction instance if
  - The federate has subscribed to the interaction class of the instance
  - The instance can be promoted (i.e., up the hierarchy) to a subscribed interaction class.

When a federate discovers an object, it learns the object class of the instance. If the federate discovers the object instance to be of object class \( X \), it will always believe the object’s type to be \( X \). If a federate subscribes to class \( X \) and not to class \( Y \), it will discover \( Y \) instances as \( X \) instances. If the federate subsequently subscribes to class \( Y \), object instances previously discovered as \( X \) instances (via promotion) will continue to
be seen as $X$ instances. Subsequently discovered instances of object class $Y$ will be discovered as instances of object class $Y$.³

5.3.4 Interaction Subscription

As with object classes, each federate subscribes to the interaction classes it wishes to receive. It is not possible to subscribe to individual parameters of an interaction class. Again, interactions are "all or nothing." As with object classes, a federate is informed about a new interaction instance if (a) the federate has subscribed to the interaction class of the instance or (b) the instance can be promoted (i.e., up the hierarchy) to a subscribed interaction class. When an interaction instance is promoted, only the parameters of the subscribed class are presented to the receiving federate.

5.3.5 Control Signals

In Figure 5.2, above, Federate #1 indicated that it was capable of producing $Y$ instances, but could only provide the attributes $\{b, e, f, g, h\}$. In that same figure, Federate #2 subscribes to attributes $\{a, b, c, d, e\}$ for object class $X$. The $Y$ instances produced by Federate #1 are discovered as $X$ instances by Federate #2.

Federate #2 is only interested in a few of the $Y$ attributes produced by Federate #1. As discussed previously, Federate #2 cannot access attribute "h" since the attribute isn't a part of class $X$. Further, Federate #2 has no interest in attributes $\{f, g\}$. Of the information Federate #1 is able to produce $Y:\{b, e, f, g, h\}$, only the information $Y:\{b, e\}$ is required – assuming Federate #2 is the only other federate in the federation.

The RTI issues control signals to indicate the information Federate #1 should produce. By default, a federate should refrain from producing object updates unless the Local RTI Component (LRC) has indicated that a consumer exists. If Federate #1 is first on the scene (i.e., there are no consumers), it will never be signaled to begin registering $Y$ instance information.

Once Federate #2 arrives, the LRC will indicate to Federate #1 that it should register any instances of object class $Y$ with the federation execution and it should start providing updates for $Y:\{b, e\}$. If Federate #2 goes away, Federate #1 will be told to stop registering instances of object class $Y$ and to stop providing updates for $Y:\{b, e\}$.

³ Rediscovery of an object instance can be forced using the `RTIambassador::localDeleteObjectInstance()` service. After object class $Y$ was subscribed to, a federate could “locally delete” all instances of object class $X$ to rediscover the objects based on the federate’s new subscriptions.
Each LRC informs its federate (via callbacks) which object attributes and which interactions to start or stop producing based on consumer demand. Each federate’s Simulation Object Model (SOM) will identify the extent to which the federate does or does not make use of the control signals provided by the LRC.

5.4 Object Publication and Subscription

Each federate is responsible for identifying its publication and subscription interests to the RTI LRC using the RTIambassador methods subscribeObjectClassAttributes() and publishObjectClass(). The interaction diagram shown in Figure 5.9, Object Publication and Subscription, illustrates the procedure for building the information required to use these methods.

The publish and subscribe methods both require an RTI::ObjectClassHandle and an RTI::AttributeHandleSet. The LRC has an internal (numeric) representation for object classes, object class attributes, interaction classes and interaction class parameter string representations that appear in the FED file. RTIambassador methods like getObjectClassHandle() and getAttributeHandle() translate character descriptions into LRC handles.

The (abstract) class RTI::AttributeHandleSet identifies a set of attributes – e.g., \{a, b, c, d\}. To express interest in publishing or subscribing to an object class, the following steps are required.

For each object class to be published:

a) Obtain the handle for the current object class.
b) Create a free-store allocated AttributeHandleSet using the static create() method in the class AttributeHandleSetFactory.
c) For each attribute the federate can publish:
   i) Obtain the handle for the current attribute.
   ii) Add the handle to the AttributeHandleSet
d) Publish | Subscribe the AttributeHandleSet for the object class.

5.5 Throttling Publications

The LRC signals a federate to start or stop registering object instances for all published object classes and generating interactions for all published interaction classes.

5.6 Object Declaration

Excerpts from helloWorld, illustrating publication and subscription using RTIv1.3, are included in Part 5 of the Lab Notebook. General comments about the use of the RTI services are included below.
5.6.1 Dynamic Object Publication and Subscription

Each call to `publishObjectClass()` and `subscribeObjectClassAttributes()` for an object class replaces previous calls. The methods `unpublishObjectClass()` and `unsubscribeObjectClass()` should be called when a federate is no longer interested in any attributes of an object class.

5.7 Publishing and Subscribing Interactions

Registering publication and subscription interest in interaction classes is more straightforward than registering interest in object classes. Figure 5.10, Declaring Interactions, identifies `RTIambassador` declaration management methods interactions. Interactions are "all or nothing." Unlike object registration, you cannot specify interest in particular interaction parameters.

As with object class declaration, interaction interest can be declared dynamically. Each call to `publishInteractionClass()` and `subscribeInteractionClass()` for an interaction class replaces previous calls. The methods `unpublishInteractionClass()` and
unsubscribeInteractionClass() should be called when a federate is no longer interested in an interaction class.

5.8 Object Management

This chapter introduces the RTIAmbassador service and FederateAmbassador callback methods that support object management. Object management includes instance registration and instance updates on the object production side and instance discovery and reflection on the object consumer side. Object management also includes methods associated with sending and receiving interactions, controlling instance updates based on consumer demand, and other miscellaneous support functions.

To create an object, the federate must first have published that object class using the declaration management services of the RTI. The publishing federate then registers the object using the declaration management services. This results in a discover callback by the RTI to any subscribing federate. (See Figure 5.11.)
5.9 Registering, Discovering, and Deleting Object Instances

Figure 5.12 illustrates the interactions required to register and to discover object instances. The RTIambassador method `registerObjectInstance()` informs the Local RTI Component (LRC) that a new object instance has come into existence. The method requires the object class of the new object instance and an optional object name. The method returns an `RTI::ObjectHandle` which the LRC uses to identify the particular object instance.

Objects

- To create an object, the federate must have published that object class (declaration mgt.)
- The federate registers the new instance of the object
- To discover this object, other federates must have subscribed to that object class (declaration mgt.)
- Those federates then discover the object instance

Registration introduces an object instance to the federation. It does not, however, provide attribute values for the instance. That requires a second step.

Each and every object can be deleted by exactly one federate. Initially, the federate that creates (registers) an object has the privilege to delete the object. In Figure 5.12, the RTIambassador method `deleteObjectInstance()` is called to remove a registered object. The FederateAmbassador `removeObjectInstance()` callback informs federates that a previously discovered object no longer exists. The RTIambassador method `localDeleteObjectInstance()`

---

4 The topic Ownership Management, covered in Module 2, explores functions for giving away the privilege to delete as well as the right to update various attributes.
effectively "undiscovers" an object instance. This method does not ensure the object will be permanently undiscovered. This service is intended to be used when a federate discovers an object as an instance of an object class but would like to subscribe to object classes that extend the discovered class and then rediscover the instance based on the new subscriptions. The object instance will be rediscovered upon the next `updateAttributeValues()` invocation that meets the receiving federate’s subscriptions.

Figure 5.12: Object Management Methodology
5.10 Updating and Reflecting Object Attributes

To update one or more attributes associated with a registered object instance, a federate must prepare an RTI::AttributeHandleValuePairSet. This set is similar to the RTI::AttributeHandleSet discussed previously in Section 5.0, Declaration Management. An AttributeHandleSet, abbreviated AHS, identifies a set of attributes. An AttributeHandleValuePairSet, AHVPS, identifies a set of attributes and their values. The static function RTI::AttributeSetFactory::create() is used to construct a free-store allocated AHVPS instance.\(^5\) In Section 5.0, Declaration Management, the notation \(\{a, b, c, d\}\) was used to identify four attributes by name. The notation can be extended to accommodate attribute values – e.g., \(\{a = 5, b = \text{"Hello"}, c = 14.79821, d = -12\}\).

Attribute updates are provided for an object instance via the RTIambassador method updateAttributeValues(). The method requires an ObjectHandle, which the LRC uses to identify an object instance, an AHVPS and a descriptive character string (tag). An optional FedTime argument will have meaning if the federate is "regulating," and one or more contained attributes are TSO (see Part 4: The Role of Time, and Time Management).

Figure 5.13, Object Management Updates, illustrates the interactions required to discover and to reflect updates for external object instances. Discovery is the counterpart to registration. Reflection is the counterpart to attribute updates. The FederateAmbassador callback method discoverObjectInstance() informs the federate that a new object instance has come into existence. The method provides an object handle which will be used to identify the object for subsequent updates, etc. The method also identifies the object class of the new object instance. It's extremely important to note that the ObjectHandle is a local (numeric) representation maintained by the LRC. The same object instance is typically known by different handle values in each federate.\(^6\)

5.11 Encoding and Object Update

When producing an AHVPS, the federate is responsible for any data marshalling (encoding). The LRC knows nothing about data content. It knows the names of object classes, the names of attributes and the numeric handle representations for object classes and attributes. The following code demonstrates how an AHVPS is produced for the Country class introduced in previous chapters. Data is encoded and the length of the

---

\(^5\) AHVPS is actually an abstract class; so, the factory function produces an AHVPS descendant (implementation).

\(^6\) For instance, in a particular implementation of the RTI, Federate #4 may know an object instance as 1278443. Federate #12 may know the object instance as 956. Each federate’s LRC takes care of translating the local instance representation as required when communicating with other federates.
encoding is communicated to the LRC.\textsuperscript{7} Ultimately, the AHVPS is bound to an object instance handle in an updateAttributeValues() invocation.

5-13 Exchanging Interactions

Interactions are constructed in a similar fashion to the way attribute updates are constructed. Recall that objects persist, interactions do not. Each interaction is constructed, sent and forgotten.\textsuperscript{8} Interaction recipients receive, decode and apply the interaction. Recall that, as for objects, to send an interaction, the federate must have

\textsuperscript{7} The AHVPS actually consist of triples, not pairs. The triple is (1) the attribute handle, (2) the corresponding value and (3) the length of the encoding.

\textsuperscript{8} Interactions can be retracted. See the manual pages from the Programmer’s Guide for details.
published that interaction class (per declaration management). Also, as shown in Figure 5.14, to receive an interaction, a federate must have subscribed to that interaction class (per declaration management).

Interactions

- To send an interaction, the federate must have published that interaction class (per declaration management)

- To receive this interaction, other federates must have subscribed to that interaction class (per declaration management)

Figure 5.14: Interactions

Figure 5.15, Exchanging Interactions, lists the classes and methods involved in generating and consuming interactions. RTIambassador methods are discussed in Appendix A, RTI::RTIambassador, FederateAmbassador methods in Appendix B, RTI::FederateAmbassador, and the supporting types (e.g., ParameterHandleValuePairSet and ParameterSetFactory) in The Programmer’s Guide: Supporting Types and Classes.
5.14 Additional Object Control

Object attribute updates and interactions are conveyed between federates using one of two data transportation schemes – "reliable" and "best effort". For objects, the transportation scheme is specified at the level of individual attributes. For interactions, the transportation scheme is specified at the interaction level (i.e., not the parameter level). By default, the transportation scheme is specified per object/attribute name and per interaction name in the Federation Execution Data (FED) file.

It is possible to change the transportation scheme dynamically for one or more attributes of a specific object instance using the RTIambassador method changeAttributeTransportType(). It is possible to change the transportation scheme dynamically for interactions by class name using the RTIambassador method changeInteractionTransportType(). Figure 5.16 illustrates these functions.
5.14.1 Attribute Management

A particular federate may have created and registered a particular aircraft. If one or more federates exist and are interested in updates for this particular object instance, the LRC would issue the `turnUpdatesOnForObjectInstance()` callback to specify the particular attributes for which updates should be generated. If at some point, there was no further external interest in the aircraft, the LRC would invoke `turnUpdatesOffForObjectInstance()` – informing the federate to cease updates for this particular object instance.

---

9 These functions are companions to the declaration management callback methods `startRegistrationForObjectClass()` and `stopRegistrationForObjectClass()` (see Part 4, Declaration Management).
The federate should presume that there is no external interest in an object unless or until `turnUpdatesOn...()` is issued. Calls to `turnUpdatesOn...()` and `turnUpdatesOff...()` are cumulative. Each call to `turnUpdatesOn...()` adds to the set of attributes that should be updated. Each call to `turnUpdatesOff...()` removes attributes from the set of attributes that should be updated.

5.14.2 Enable/Disable Attribute Management

It is possible to disable the `turnUpdatesOn...()` and `turnUpdatesOff...()` callbacks. The `RTIambassador` methods `enableAttributeRelevanceAdvisorySwitch()` and disable `attributeRelevanceAdvisorySwitch()` can be used to specify whether per object instance control signals are generated or suppressed.\(^{10}\)

\(^{10}\) These methods are not shown in the accompanying interaction diagram.

Figure 5-17: Scope Interactions
5.14.3 Attribute Scopes

Prior to communicating attribute updates for a particular object instance, the LRC will (at the federate's discretion) provide the preliminary callback attributesInScope() announcing that subsequent attribute updates for the specified object and specified attribute set may be forthcoming. A subsequent attributesOutOfScope() callback would inform the federate that subsequent attribute updates for the specified object and specified attribute set would no longer be provided. These signals will be generated or suppressed based on the "attribute scope advisory switch" which is set by the RTI ambassador methods enableAttributeScopeAdvisorySwitch() and disableAttributeScopeAdvisorySwitch(). Figure 5.17 provides an interaction diagram for these methods.

Assignment

1. Continue the effort from part 4 to create a new, passive federate, the UNFederate, that collects and displays data from any running helloWorld federates.

2. The helloWorld federate is much more complicated than absolutely necessary as it serves as a template and demonstration of the structure of much more complex simulations. Try to eliminate as many unnecessary functions from the UN federate as you can to make it as simple as absolutely possible. Many of the callback routines in the federateAmbassador may be eliminated by simply not performing the actions in the UN federate that trigger unnecessary callbacks.

3. Using the HLA Course Lab Notes, Chapter 5, and the source files for helloWorld, determine the data structures used in the helloWorld Federate to store the attributes of an object of Class Country, the locations in the program where class country is published and subscribed, and where country objects are registered and discovered. If more than one function is actually involved in fulfilling one of the above actions, describe the use of each function. Repeat the above for interactions and their parameters used by the helloWorld federate.

4. Determine the mechanisms used by the helloWorld federate to control excess message traffic and object registrations, and where the corresponding calls are located in the code.

5. Determine why there are so many variants of the update and constructor functions for Class Country in the helloWorld Federate, and the purpose for each variant.
Suggested Readings


2. Read Chapters 7, and 8 of the High Level Architecture Run-Time Infrastructure Programmer’s Guide. Also look up the interface specifications for the interfaces used in this lesson in the Appendices to this Guide.