iPlant Semantic Web Services
Code Design Document

Clark & Parsia
University of Arizona

May 20, 2010

Abstract

Contents

Todo list

1 Introduction

This document describes the code design for SSWAP Java API and its implementation.

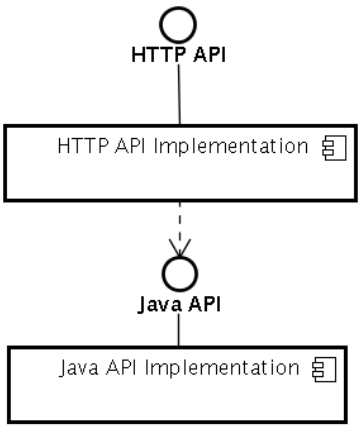

Figure 1: SSWAP Components

2 Structural overview

Figure 1 presents the the main components of SSWAP and their typical deployment. The HTTP API Implementation has three purposes: provide a network-enabled layer, allow access to SSWAP to non-Java code by allowing to treat SSWAP as a webservice, and also to provide a simplified API (for less advanced users). Java API Implementation is the more advanced API that allows full manipulation of SSWAP objects and fine grained control about the service invocation and reasoning. Within the same JVM, HTTP API is always the upper layer; that is, it invokes Java API. On the other hand, Java API may invoke other remote HTTP APIs.

A typical scenario of deployment of SSWAP components is shown on Figure 2. It shows a client machine and a provider's (server) machine. Within the client, there are three types of client code accessing SSWAP: two clients accessing using the simple HTTP API, and one accessing Java API directly. On the provider's side, there is also a "stack" of HTTP API Implementation and underlying Java API. At the bottom of the stack there is the code actually implementing the service. (Technically, the code implementing the service may be on another machine than the HTTP API and Java API accepting the calls to SSWAP Provider, if a remote provider invocation API is defined.)

2.1 Details of HTTP API

To be designed.

2.2 Details of Java API

As shown on Figure 2 the Java API component consists of a defined API (consisting mostly of a set of Java interfaces), and the implementation. The implementation in turn encapsulates Jena models (for storing and manipulating RDF data) and Pellet OWL reasoner (using its Jena interface).

Figure 3 shows the overall class hierarchy of the classes and interfaces in the API (top) and the implementation (bottom). The API consists of mostly interfaces, with the exception of SSWAP class, which is a

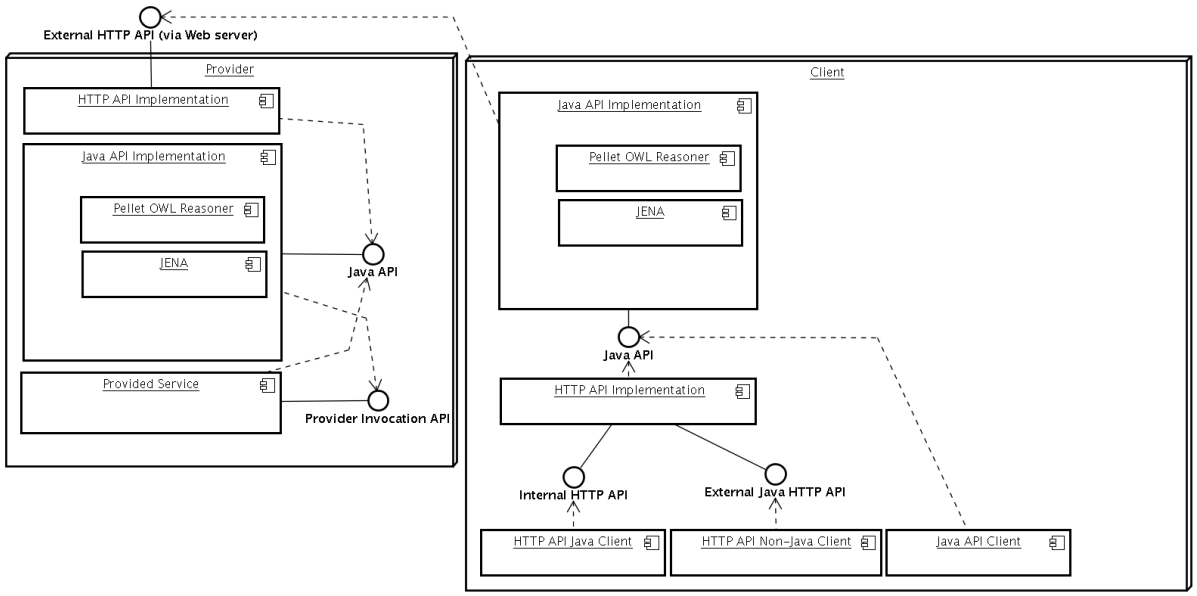

Figure 2: Typical SSWAP deployment with a client (right) and a provider (left)

factory class, and it allows the users to create objects implementing these interfaces. The actual implementations are interchangeable because they can be plugged in using the `APIProvider` interface in the SPI (Service Provider Interface) package.

The class hierarchy in the implementation of the API generally follows the interface hierarchy in the API (there are a few additional intermediate classes in the hierarchy like `EmpireGeneratedNodeImpl`, and the classes implemeting API interfaces are named in the following way: without the `SSWAP` prefix (to reduce the name length) and with `Impl` suffix. For example, the class implementing `SSWAPElement` interface is `ElementImpl`.

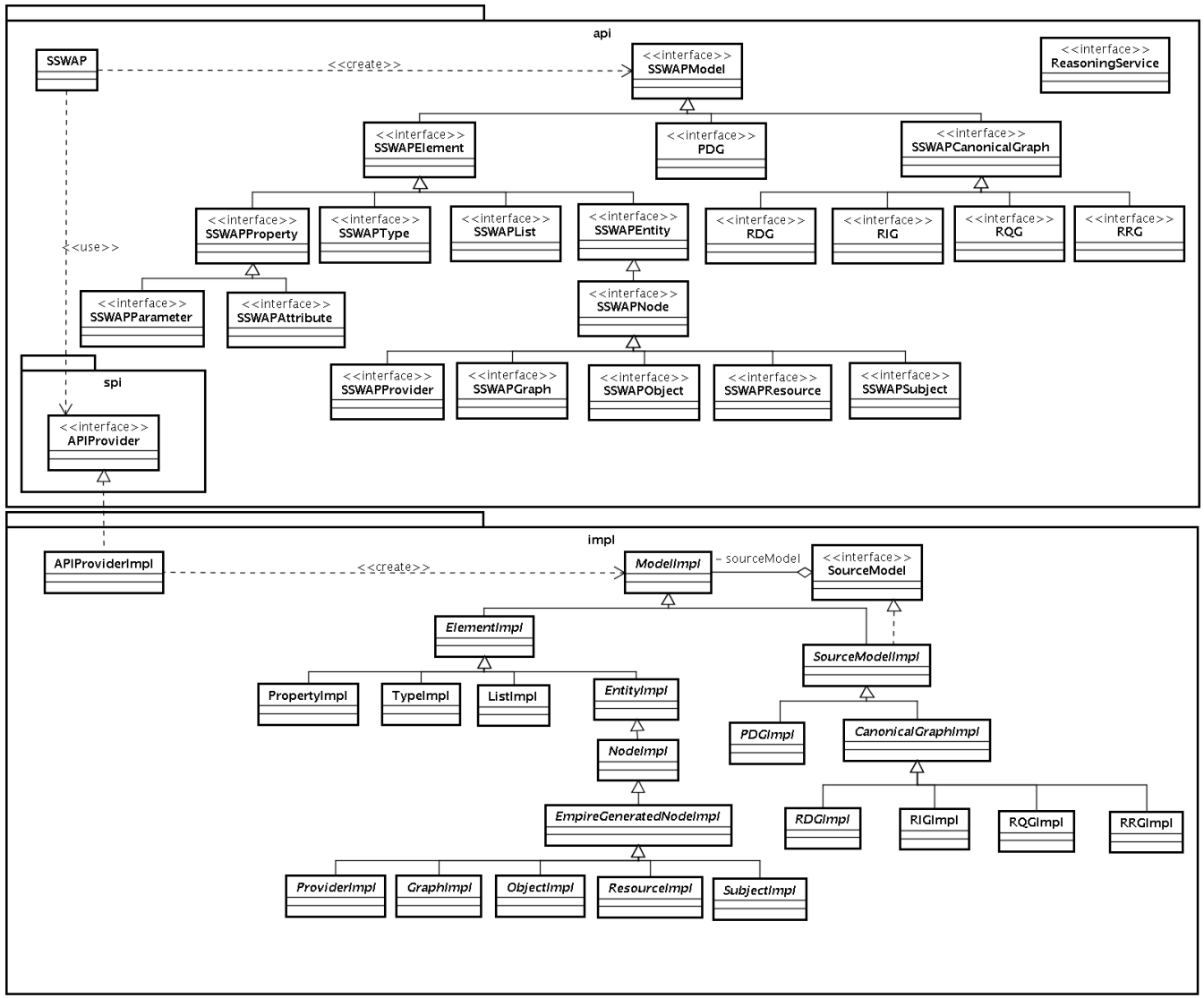

Figure 3: API and its implementation class diagram. To avoid excessive clutter, the realization relations between interfaces and classes implementing them are not shown.

2.2.1 Storing RDF data and reasoning services

Technologies used Java API implementation uses two technologies for handling RDF data: Jena[1] and Empire[2].

Jena is an open-source Semantic Web Framework initiated originally by HP Labs Semantic Web Programme. In Java API it is mostly used to parse RDF data sources and provide object-oriented representation for RDF statements. These RDF statements are stored in memory in Jena `Models`.

Empire is an open-source implementation of the Java Persistence Architecture (JPA) for RDF and the Semantic Web. Empire simplifies operating on the underlying RDF data by providing an automated mapping between an RDF data source (e.g., a Jena Model) and a Java object. As the result, it encapsulates the process of reading and writing of RDF data manually and copying it into fields of Java objects. Empire uses specifically annotated Java interfaces and abstract classes, and provides concrete implementations of them at run-time, which hide the RDF manipulation. (For example, if a class contains a field that will be read from RDF, the getters and setters for the field will initially be declared abstract, and Empire will provide an actual implementation of these methods.)

In Java API Implementation many classes are abstract (their names are italicized on Figure 3), most of them because of the abstract methods provided by Empire at run-time. Most important classes generated by Empire inherit from `EmpireGeneratedNodeImpl` and they represent core SSWAP concepts like SSWAP Graph, Subject, or Object.

Storing RDF data in `SourceModels` The SSWAP Java API provides a logical view to the user of the API that every object (`SSWAPModel`) contains its own set of RDF data (hence the word "model" in that top interface of the SSWAP hierarchy), and the models can be nested (e.g., `RDG`, which represents the whole RDG graph, contains all the RDF statements from that graph, and `SSWAPResource`, which is contained in that RDG, contains only the RDF statements related to that resource). However, actual implementation does not make every `ModelImpl` (corresponding to `SSWAPModel`) contain the RDF data because of excessive overhead. Instead, the actual statements are contained in objects that implement `SourceModel` interface, and all the other `ModelImpls` delegate that task to their `SourceModel`. Currently, the only objects that implement `SourceModel` interface are the ones that represent an RDF document in SSWAP (i.e., PDG and all canonical graphs like RDG, RIG, RQG, RRG). These classes implement `SourceModel` indirectly by inheriting from `SourceModelImpl`.

Figure 4 shows the part of the class and interface hierarchy of the implementation related to `SourceModels`. Typically, every `ModelImpl` has a reference to a one `SourceModel` (with the exception of the situation where the model has not been dereferenced yet; that is, populated with RDF data; in such a situation, there is no reference to the `SourceModel`). (Since `SourceModelImpl` is also a `ModelImpl`, it has a reference to itself, as its source model.) A source model contains a reference to an Empire `EntityManager` that in turn owns the Jena `Model` (for convenience, a source model knows the reference to the Jena `Model` too).

Reasoning service Since `SourceModels` have direct access to the data, they are a logical place to provide a reasoning service in SSWAP. A reasoning service (provided by `ReasoningServiceImpl`) is initial-

[1] http://jena.sourceforge.net
[2] http://github.com/clarkparsia/Empire

ized lazily (only when requested for the first time) because of high computational overhead – it requires computation of Jena `OntModel` by classifying the data in the original model.

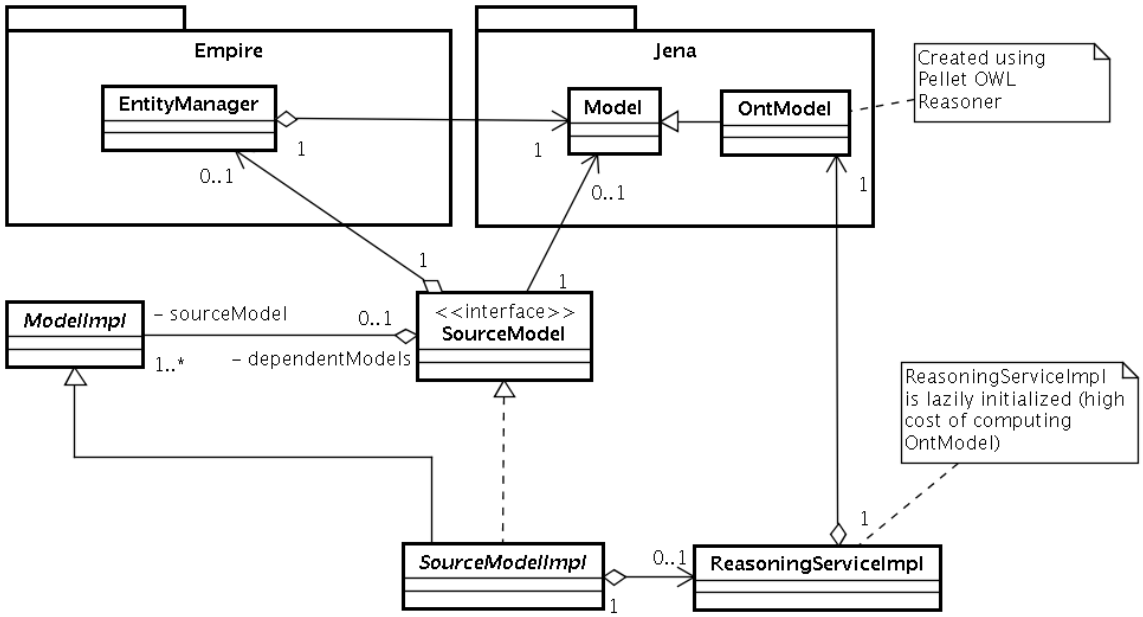

Figure 4: Relation of Jena Models (containing RDF data) and reasoning services to classes in API implementation.

3 Sequence diagrams for use-cases

This section describes the code interaction while executing the most common use cases.

3.1 Publish provider (PDG)

Figure 5 shows the interactions among the objects when the user creates a new PDG using Java API with the purpose of putting the necessary data about a SSWAP Provider on the web. The use case starts with a call to `SSWAP` class, which is the factory class of the API. Next, `SSWAP` delegates the task to the implementation (`APIProviderImpl`), and creates an object implementing the `PDG` interface (`PDGImpl`). Since the information about the provider is stored in a `SSWAPProvider` within a PDG, an implementation for the provider (`ProviderImpl`) is also created. Moreover, since the underlying RDF data for `ProviderImpl` is stored in `PDGImpl`, `setSourceModel(pdg)` is called (this also adds provider object as a dependent model to the PDG). Next, all the necessary setter methods are invoked to initialize all the data. Finally, the user can invoke a method to create a zip archive with the necessary files and directory structure that can be uploaded to a server.

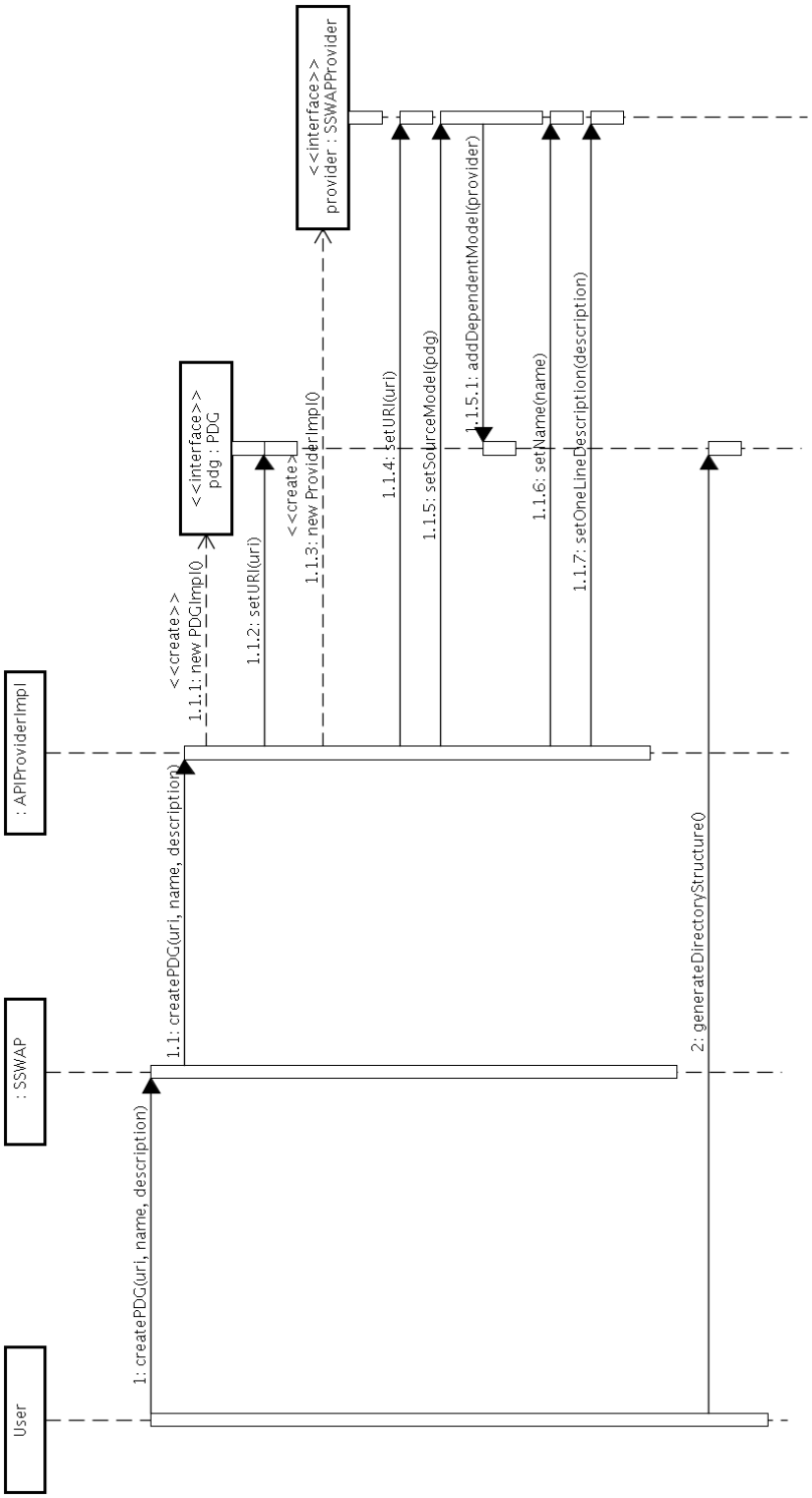

Figure 5: Publish provider sequence diagram

3.2 Publish resource/service (RDG)

Figure 6 shows the use case of publishing a service (by publishing its RDG). The beginning of the interaction is analogical to the use case of publishing a PDG (described in Section 3.1 and on Figure 5). However, after creation of the implementation of `SSWAPResource` (step 1: `createRDG` and all subsequent calls), the user can create mappings within a resource. In the example shown, the user creates a one-to-many mapping; that is, a service with one parameter and multiple results (one `SSWAPSubject` and multiple `SSWAPObjects` – on the diagram there are shown two objects). (Note: since the subjects and objects are dependent on the RDG, in the implementation are `setSourceModel(rdg)` calls for each of these objects. The diagram does not show these calls to avoid unnecessary clutter.) The end of the use case is again similar to the previous use case – generating the files and the directory structure (to a zip archive).

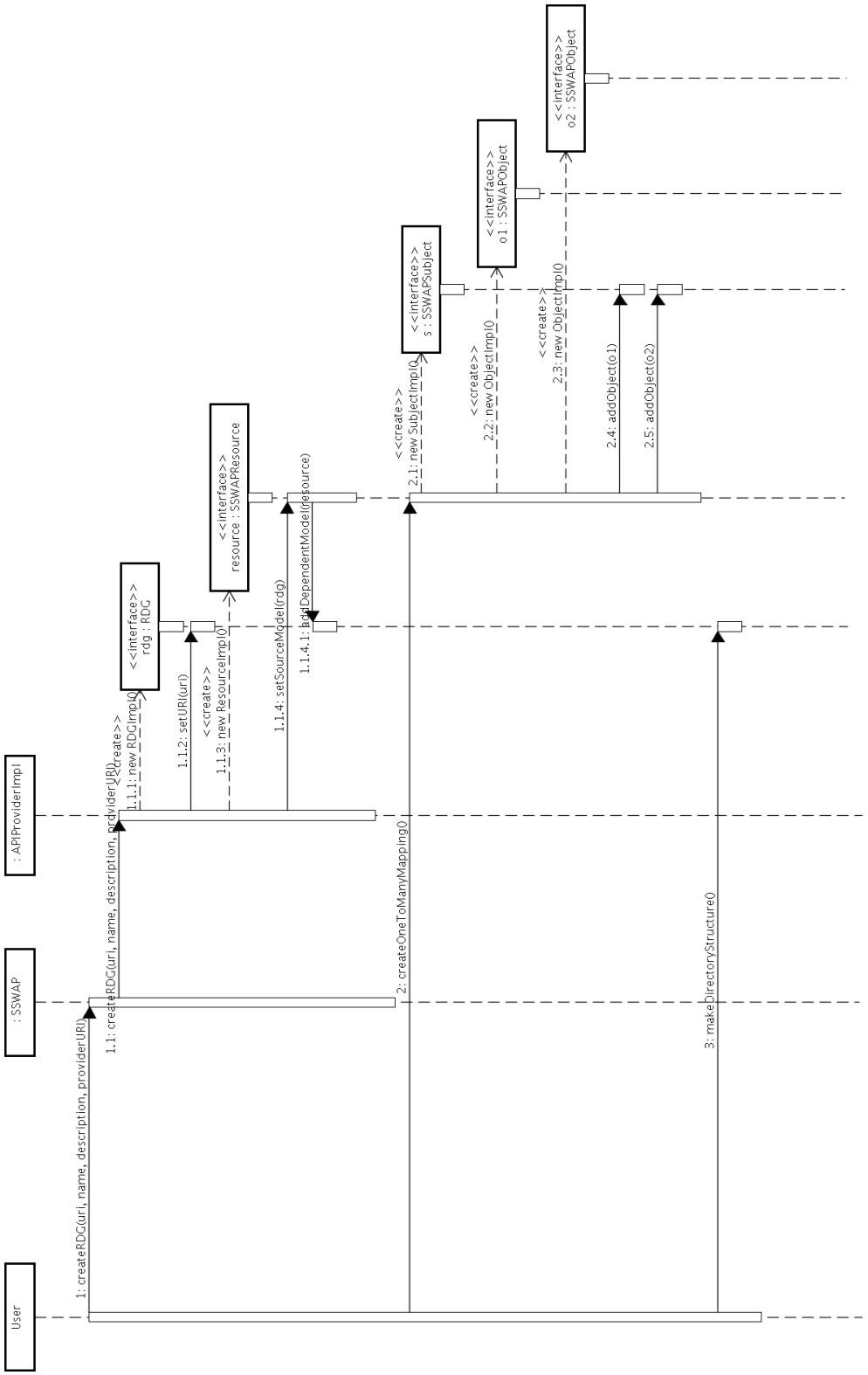

Figure 6: Publish resource/service sequence diagram

3.3 Retrieving a resource from RDG

Figure 7 shows a sequence of calls that occurs when a user wants to create an `RDG` object based on existing (published) RDG document, and then to retrieve a `SSWAPResource` from this document.

The first call (`getRDG()`) is forwarded to the implementation, and it creates an empty (undereferenced) object. (An undereferenced object does not have any Jena model or Empire `EntityManager` associated with it yet.) The process of creation involves calling Empire's `InstanceGenerator` (step 1.1.1.1) because `RDGImpl` class is an abstract class. (It contains Empire-annotated abstract methods whose implementation is provided by `InstanceGenerator`.)

The second user's call (`dereference()`; step 2) causes the actual retrieval of data from the Internet. `RDGImpl` creates an Empire `EntityManager` for that purpose, which in turn creates a Jena model, and populates it with RDF data. Having populated the model, `refresh()` methods are called (steps 2.2 and 2.2.1) to force Empire to transfer the data from Jena model to RDG.

The third user's call (step 3) retrieves the `SSWAPResource` object from the RDG. Since at this step, the object implementing `SSWAPResource` has not yet been created (lazy initialization), Empire is asked to create this object. In step 3.1, `getDependentObject` method initiates the whole procedure after having received the class (`ResourceImpl`) that Empire should complete. (Similarly to `RDGImpl`, `ResourceImpl` is an abstract class, and Empire has to provide the missing methods.) Before the actual `ResourceImpl` object is created (in step 3.1.1.1.2), Empire queries the underlying Jena source to ensure that there exists RDF data (i.e., an individual of `sswap:Resource` type) to back the newly created object (steps 3.1.1.1 and 3.1.1.1.1).

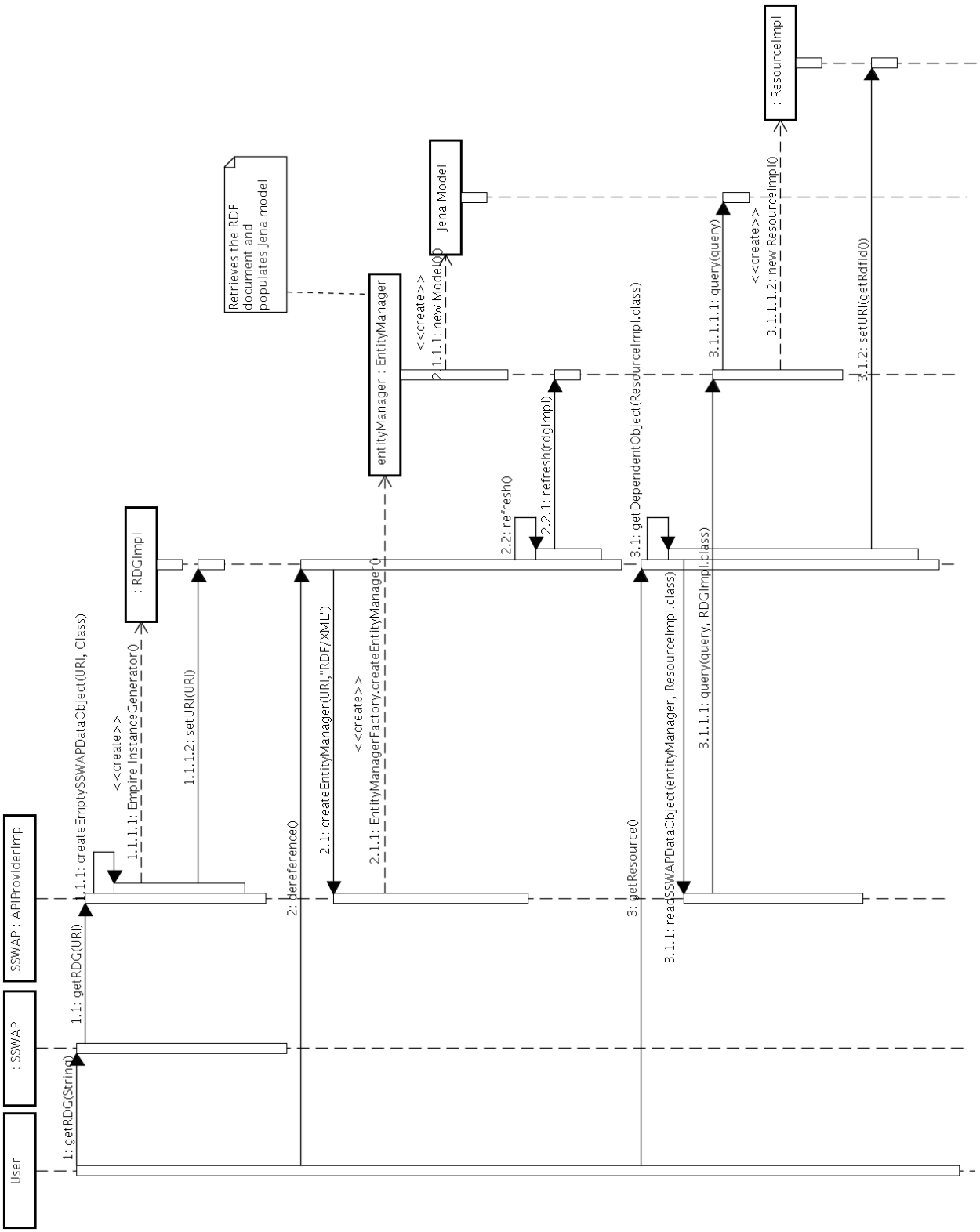

Figure 7: Sequence diagram showing retrieving a resource from an RDG

3.4 Service invocation

Scenario presented on Figure 8 shows the actions occurring on the server side when a RIG from the client is received. (The actor in this sequence is the code within HTTP API, which is denoted on the diagram as *RIG HTTP Handler*.)

The scenario begins with creation of the object corresponding to the serialized version of the RIG data – `RIGImpl` (steps 1 and 1.1). When a RIG object is created, also all types defined in that document are cached, and appropriate objects implementing `SSWAPType` are created. (These objects are of `TypeImpl`.) At the end (step 1.2), `validate()` is called to verify that this canonical graph adheres to SSWAP standard.

The second step involves calling `doClosure()`, which gathers additional data about the unknown concepts. The data can be retrieved using one of the core SSWAP assumptions – the ability to dereference the URI of any concept. The retrieval is continued until a desired depth level of search is reached, a time limit is exceeded or no more new concepts can be added. During the closure computation new `SSWAPTypes` can be discovered (step 2.1), and (similarly to step 1.1.1) corresponding `TypeImpl` objects will be created and cached.

Later (step 3), the application can verify whether the basic subject-object pattern matches its expectation for the service (e.g., the service may expect a single subject, and is supposed to produce multiple object – a one-to-many mapping). If the pattern matches the expectation of the service, the next crucial step is to ensure that the type of the RIG's subject (parameter or input data for the service sent by the client) is a subclass of the type required by the service (denoted as `neededType` on Figure 8 in step 5). (It is possible and legal for a client to send as a subject a class that is a more specialized version than the service requires.) The retrieval of all types of the subject occurs in step 6, and then for each of these types it is verified whether that type (`subjectType`) is a subclass of `neededType`. (It is only necessary for one `subjectType` to be a subclass of the required type.)

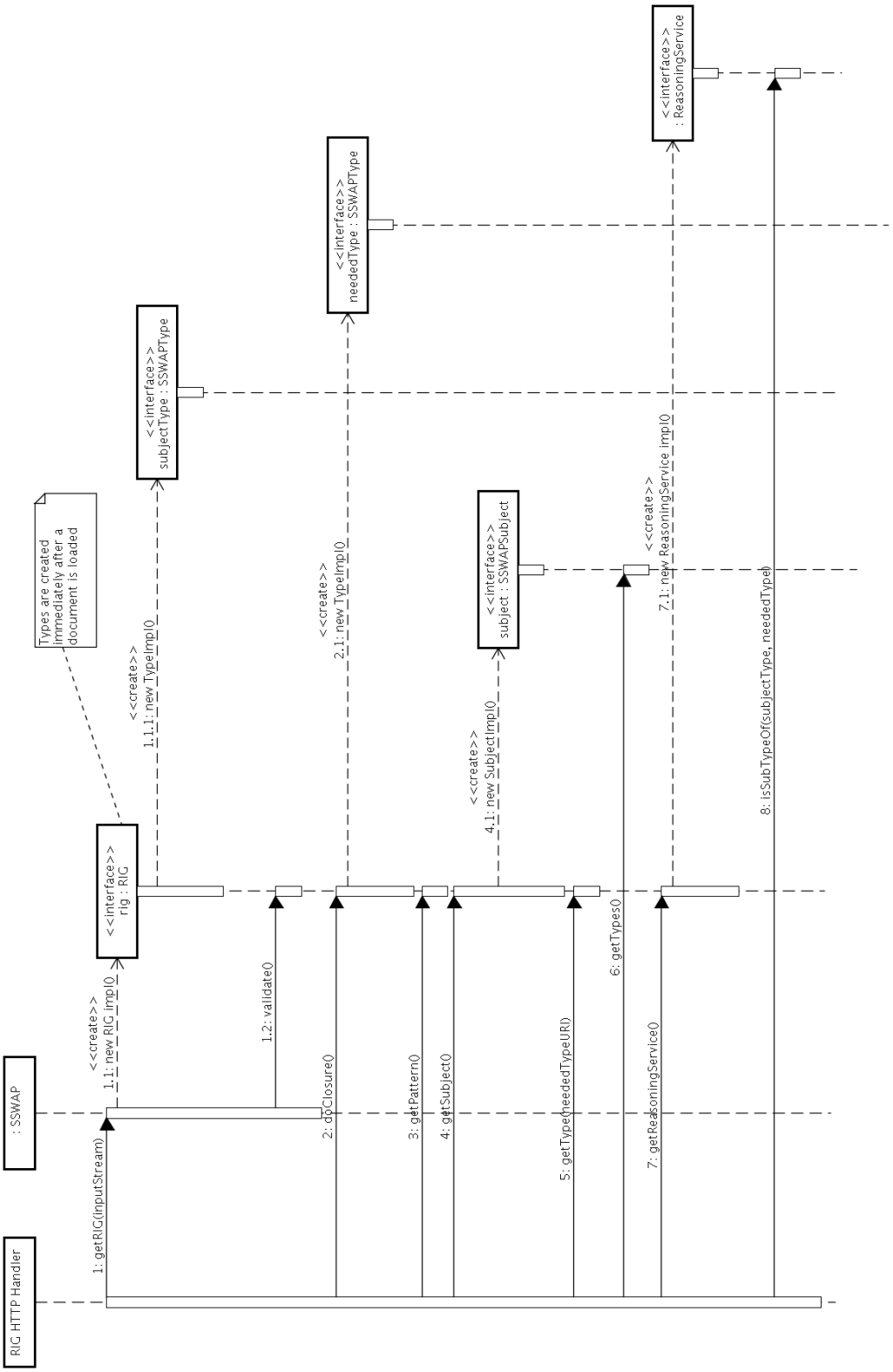

Figure 8: Sequence diagram showing calls during a service invocation (provider side)

4 Acronyms

This chapter lists all the acronyms used in this document:

DS Discovery Server

PDG Provider Description Graph

RDG Resource Description Graph

RIG Resource Invokation Graph

RRG Resource Response Graph

RQG Resource Query Graph