OMTC : Open Machine Translation Core
An open API standard for Machine Translation Systems

Version: 0.6.1-DRAFT

Ian Johnson
Capita Translation and Interpreting

February 26, 2013

Abstract

Open Machine Translation Core (OMTC) is an open standard that defines an application programming interface (API) for machine translation systems (MT). The API defined is a service interface which can be used to underpin any type of MT system: web-based, traditional client-server, or standalone single process applications. It consists of components, some of which are optional, which allow application programmers to implement the core of MT systems in such a way as they "look" consistent. This makes MT applications easier to develop since disparate MT products start to "look" the same.

MT applications provide significant portions of functionality to OMTC that allow aspects of the standard API to be customised for the application being developed. However, applications are not tied to the API allowing existing MT systems to be re-factored to use OMTC. As well as new MT systems to be easily and quickly developed using the standard API.

This standard API was developed as part of the MosesCore project sponsored by the European Commission's Seveth Framework Programme. For more information on the Seventh Framework Programme please see `http://cordis.europa.eu/fp7/home_en.html`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Open Machine Translation Core (OMTC) is an application programming interface (API) for machine translation (MT) systems. It defines a *service interface* for an MT system. It is intended that this API makes different flavours of MT system "look" the same, so that they can be used in an MT system consistently. This interface is *not* one with which disparate MT systems can immediately communicate, rather it provides a framework in which an MT system can reside. This leaves application concerns, e.g., inter-process communication, security, and payment, to be implemented around the API.

The approach taken in OMTC separates the concerns and responsibilities into their correct "silos" and attempts to provide a highly cohesive design that, when implemented, is not tightly coupled. Since the architecture of a machine translation (MT) system cannot be predicted, defining a service interface keeps the "shape" of compliant applications consistent without tying down the application programmer to particular technologies or protocols.

A note on interoperability: this API does not intend to address the fundamental problems around the exchange of translation memories. This has been identified as a significant problem for language service providers (LSPs) and could be losing them 30% of their business. Inspite of this, the approach this API has taken to this issue is to avoid it altogether. MT systems could be implemented using any, all or none of the standard exchange formats, so we do not try and second guess the application programmer. It is assumed that support for schema conversion from one standard to another is written as part of the implementation of the MT system.

Alternatively, libraries for converting from schema to schema can be developed but this is an application concern since it is the application that defines which schemas are to be supported. Moreover, it is the application which defines which schemas can be converted to other schemas. The serialisation of such exhange formats is be dependent on the transport chosen. Since transport is defined by the application, then so is the serialisation.

1.2 Architecture

The API is separated into a number of packages. These are:

- *Resources*: An abstract representation of things that will be used by users of an MT system, i.e., documents, translation memories, glossaries and MT engines,
- *Sessions and session negotiation*: An abstract session implementation to allow users to upload and download resource to and from an MT system. Negotiation ensures that both client and server in an MT system meet each others feature expectations,
- *Authorisation integration*: Since users may need authorisation to execution certain operations integration with the application defined authorisation system is support by this API,
- *Task scheduling*: Machine translation uses computationally expensive algorithms. If multi-user MT systems are to be constructed a fair way of scheduling and executing computationally expensive software is required. The scheduling API is an abstract representation of a detached execution environment which can map to technologies which can efficiently utilise the computing resources available to an MT system. These technologies can be anything from native threading to grid or cluster computing infrastructure,
- *MT engines*: The API defines an abstract representation of entities that performance machine translation. The MT engine package should support *any* flavour of MT engine, and optional functionality can be mixed in so that application programmers can tailor their implementations to their MT systems. The optional behaviour is often computational expensive and the task scheduling API is used to submit these tasks into the scheduling implementation, and

- *Translators*: The abstract translator API provides an interface to submit computationally expensive translation tasks into the task scheduling API.

1.3 Notational Coventions

In this document, any class name, method name, or identifier associated with the API definition shall be represented in a `fixed-width` font.

All diagrams in this standard that defined API structure are shown using the Unified Modeling Language v1.x[1] (UML).

The key words ”MUST”, ”MUST NOT”, ”REQUIRED”, ”SHALL”, ”SHALL NOT”, ”SHOULD”, ”SHOULD NOT”, ”RECOMMENDED”, ”MAY”, and ”OPTIONAL” in this document are to be interpreted as described in RFC 2119[2].

[1]The UML v1.x standard can be found at `http://www.uml.org/`.
[2]RFC 2119 is available at `http://www.ietf.org/rfc/rfc2119.txt`.

Chapter 2

Implementation

2.1 Language Bindings

This standard has been defined using a language-agnostic representation. The appeal of UML is that is can be taken and implemented in many object-oriented (OO) and many non-OO languages without loosing the central concepts of the API. With this in mind, the implementation:

- MUST adhere to the standard, or its intention, as closely as the implementation language will allow,
- SHOULD map collection types to implementation language collection types that are more performant than their equivalents to the ones specified in this standard, and
- MAY implement interfaces, which are used commonly in the standard definition, as classes, or structures.

2.2 Namespace

In order to encapsulate the interfaces, classes and enumerations defined in this standard a namespace is to be used. This will avoid conflicts with other frameworks and libraries used in an MT system. The namespace that SHALL be used is `OMTC`. The namespace MAY be changed to fit with a specific language binding, e.g., for a Java binding the root package name could be `com.mammon.omtc`. However, OMTC MUST be included in a namespace definition.

2.3 Multithreading

An implementation of this standard SHOULD ensure that the implementation supports concurrency and re-entrantcy. The level of multithread safety SHALL be documented along with the implementation. All users of the implementation should be aware of the limitations of the abstract implementation. The application code they shall provide to the API shall be required to be "aware" of the level of multithreaded safety supported by the API implementation.

2.4 Reference Implementation

There is a freely available reference implementation of this API which has been released under the LGPL v3.0[1] software license. It is hosted on GitHub at `git://github.com/ianj-als/omtc.git`.

The reference implementation is written in Java and is compliant with Java v1.7. It is built using Apache Maven v2.2.1[2] or newer, and can be imported into most common IDE platforms using the `pom.xml` file.

To build the reference implementation clone, or fork and clone the GitHub repository, and build using the following commands,

```
$ git clone git://github.com/ianj-als/omtc.git # Or your fork
$ cd omtc
$ mvn install
```

Once installation, to your local Maven repository, has completed the latest OMTC implementation can be made available to other Maven projects by adding the following dependency to your `pom.xml` files:

```
<dependency>
    <groupId>com.capitati.omtc</groupId>
    <artifactId>omtc-core</artifactId>
    <version>[1.0,)</version>
</dependency>
```

Any bugs, enhancements or comments regarding the reference implementation should be directed to ian.johnson@capita-ti.com.

[1] See the GNU Lesser General Public License at `http://www.gnu.org/copyleft/lesser.html`.
[2] See `http://maven.apache.org/` for downloads and documentation.

Chapter 3

Resources

A resource is an object that is provided or constructed by a user action for use in the MT system. Examples of resources can be:

- Documents,
- Translation memories,
- Glossaries, or
- MT engines.

There are two kinds of resources defined by this standard:

1. *Primary resources*: any resource that has been constructed externally and uploaded, in some way, for use in the MT system. Examples of these resources are: a file, a translation memory (TM), a glossary etc. It is recommended that these resources be persisted so that future sessions may use them. Primary resources shall also be immutable, i.e., if a resource's content is to be altered it is a new resource.
2. *Derived resources*: these resources are constructed using their primary counterparts either as a conglomeration or a separate entity is created, e.g., when training a SMT engine using a translation memory (a primary resource) to create a derived resource: the engine itself.

Figure 3.1 shows the resource interface and its relationship with the primary and derived resources.

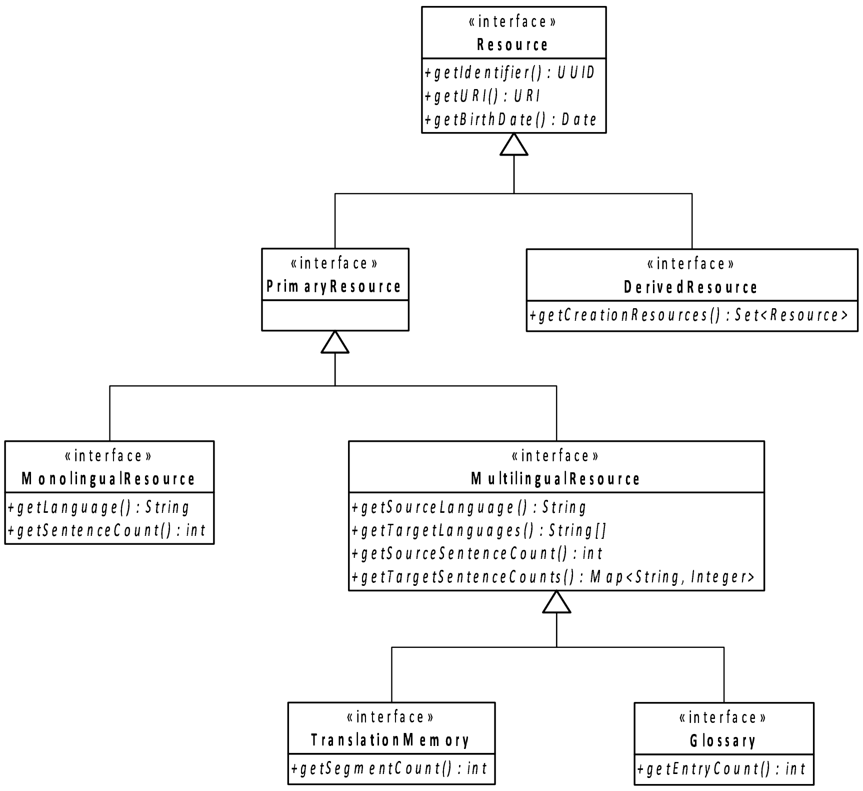

Figure 3.1: Resources interface diagram.

The primary key for any resource is its identifier, a *universally unique identifier*[1] (UUID) typed value (see `Resource::getIdentifier()`). The implementation of the UUID is implementation defined but SHALL be consistent within the MT system. When communicating with other MT systems resource identifiers SHALL be converted from one identifier space to the other: this is the responsibility of the implementer. The URI of the resource provides its location in an MT system (see `Resource::getURI()`). The scheme of the URI SHALL specify the mechanism by which the resource can be reached. Again, this is application defined but SHALL be consistent within the MT system, but implementers are free to define their own resource schemes if needed. The meaning of a resource's *birth date* (see `Resource::getBirthDate()`) is implementation defined but SHOULD be populated with some date/time stamp which is consistent within the application. A resource's birth date can be time zoned if required but conversion of time zoned timestamps MAY require conversion if the MT system is used globally.

[1]See RFC4122 at `http://tools.ietf.org/html/rfc4122`

Common resources that are used in MT system are monolingual and multilingual resources, and are modeled as the interfaces `MonolingualResource` and `MultilingualResource` respectively. The monolingual interface provides two "getters" for retrieving a, consistent and application defined, language code for the resource and the number of sentences contained in the resource: `MonolingualResource::getLanguage()` and `MonolingualResource::getSentenceCount()` respectively. The multilingual resource representation provides methods to retrieve:

- Source language: this is an application defined string whose format SHOULD be consistent throughout the application (see `MultilingualResource::getSourceLanguage()`),
- Target languages: since many target language can exist in a translation resource this is modelled as an array of strings whose format SHOULD be consistent throughout the application (see `MultilingualResource::getTargetLanguages()`),
- Source language sentence count: the number of sentence in the source language (see `MultilingualResource::getSourceSentenceCount()`), and
- Target language sentence counts: the sentence counts for all the target languages in the multilingual resource. This is modelled as a map whose keys are the target language strings, as retrieved by `MultilingualResource::getTargetLanguages()`, and their values the target language sentence count (see `MultilingualResource::getTargetSentenceCounts()`).

Two common multilingual resources are translation memories and glossaries. In the light of this, two interfaces have been defined to model these resources: `TranslationMemory` and `Glossary`. These interfaces define how to retrieve the number of segments (see `TranslationMemory::getSegmentCount()`) and entries respectively (see `Glossary::getEntryCount()`).

Derived resources add the ability to store their provenance. The resources that where used to construct the derived resource SHOULD be recorded, not only, for referential integrity but for ensuring repeatable builds of the derived resource can be done by others. Recall that *all* resources are immutable. The

`DerivedResource::getCreationResources()` method retrieves this information.

It is expected that application programmers will use these interfaces to mixin resource identity and behaviour with their representations of files, MT engines etc. If these objects are being persisted then it MAY be necessary to use a data access object if any of the resource information is to be derived from existing persistent data.

Chapter 4

Sessions, Negotiation and Authorisation

In order for users to be able to use the MT service the API needs an idea of *session*. A session is the period in which a user will interact with the MT service. The MT service application may need to acquire the identity of users, whilst other implementations may not. The MT API therefore needs to supports both user identity and anonymity. Moreover, clients to the MT service will support certain exchange formats, and expect certain features from the application. A *session negotiation* is defined in the API in order that both client and server can ascertain if, once the session is set up, their expectations of each other is correct. If the client cannot support the server's requests, or visa versa, then the session should be torn down. However, this decision is delegated to the application.

4.1 Sessions

The session class diagram is shown in Figure 4.1.

The API provides interfaces and implementations for sessions that require user identity and anonymity. The anonymous user classes are: `Session` and `AbstractSession`. These classes support actions with *resources* that *any* user would require. Resources are described in Chapter 3. The session actions are:

- Uploading a resource – a session making files and other resources available to it or other sessions in the MT application via some transfer mechanism,
- Downloading a resource – a session retrieving resources uploaded either in

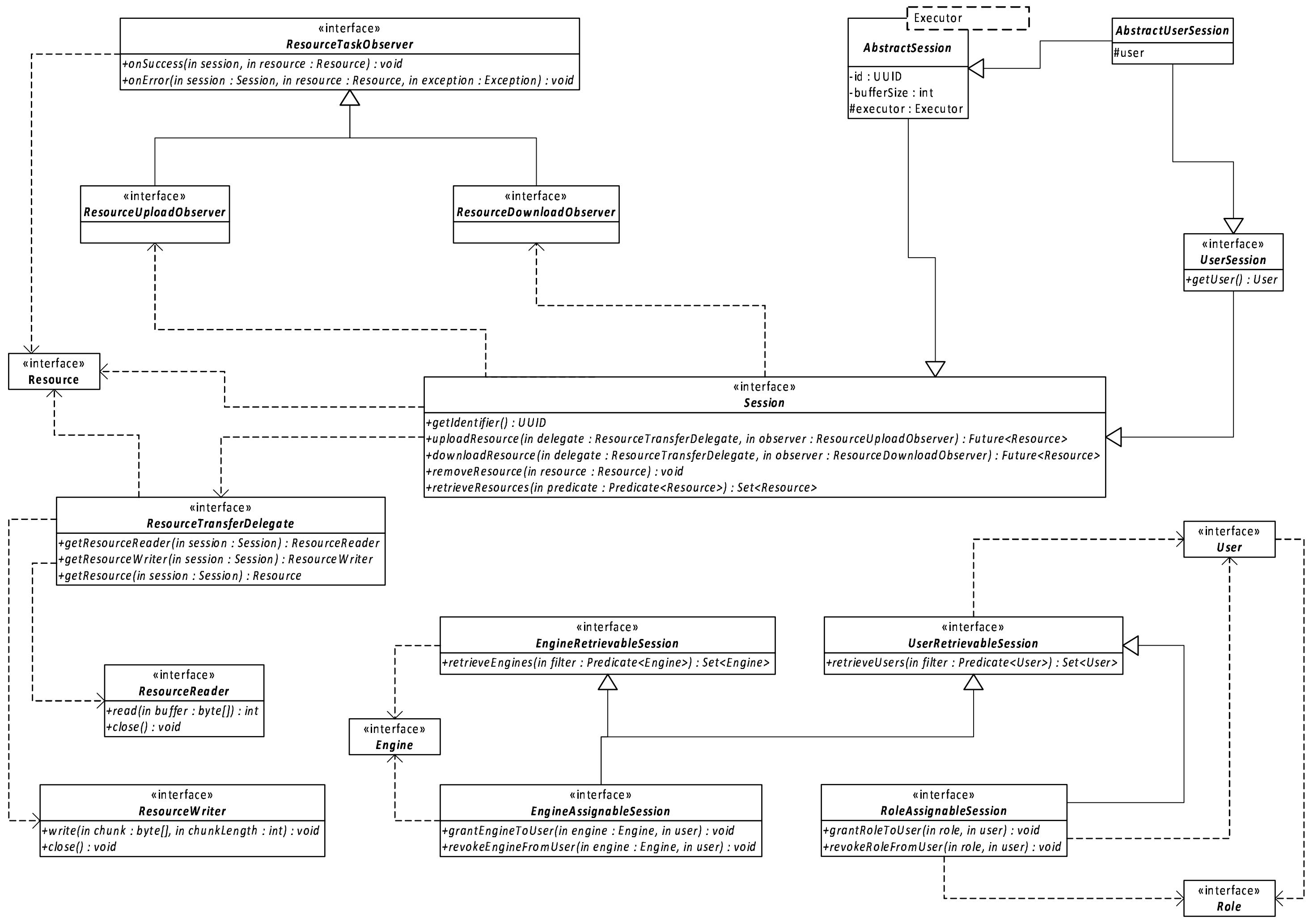
«interface»
ResourceTaskObserver
+onSuccess(in session, in resource : Resource) : void
+onError(in session : Session, in resource : Resource, in exception : Exception) : void
«interface»
ResourceUploadObserver
«interface»
ResourceDownloadObserver
Executor
AbstractSession
-id : UUID
-bufferSize : int
#executor : Executor
AbstractUserSession
#user
«interface»
UserSession
+getUser() : User
«interface»
Resource
«interface»
Session
+getIdentifier() : UUID
+uploadResource(in delegate : ResourceTransferDelegate, in observer : ResourceUploadObserver) : Future<Resource>
+downloadResource(in delegate : ResourceTransferDelegate, in observer : ResourceDownloadObserver) : Future<Resource>
+removeResource(in resource : Resource) : void
+retrieveResources(in predicate : Predicate<Resource>) : Set<Resource>
«interface»
ResourceTransferDelegate
+getResourceReader(in session : Session) : ResourceReader
+getResourceWriter(in session : Session) : ResourceWriter
+getResource(in session : Session) : Resource
«interface»
User
«interface»
EngineRetrievableSession
+retrieveEngines(in filter : Predicate<Engine>) : Set<Engine>
«interface»
UserRetrievableSession
+retrieveUsers(in filter : Predicate<User>) : Set<User>
«interface»
ResourceReader
+read(in buffer : byte[]) : int
+close() : void
«interface»
Engine
«interface»
ResourceWriter
+write(in chunk : byte[], in chunkLength : int) : void
+close() : void
«interface»
EngineAssignableSession
+grantEngineToUser(in engine : Engine, in user) : void
+revokeEngineFromUser(in engine : Engine, in user) : void
«interface»
RoleAssignableSession
+grantRoleToUser(in role, in user) : void
+revokeRoleFromUser(in role, in user) : void
«interface»
Role

Figure 4.1: Session interface diagram.

the current or previous sessions,

- Listing session owned resources – retrieving information about resources that can be used by a session, and
- Removing a resource – deleting resources, owned by the session or other sessions, that are no longer required.

Uploading and downloading resources could, potentially, take a "long" time so the execution of these operations can be detached from the invoking thread in the provided implementation. Futures, and observers (see `ResourceUploadObserver` and `ResourceDownloadObserver` shown in Figure 4.1) are used in order that the caller may block waiting for the resource to up- or download, or be asynchronously notified via the observer once the operation has completed. This gives implementations the choice of synchronous or asynchronous notifications. Also, the transfer of the resource is delegated to the application since the transport layer is chosen by the implementation. A resource's location is determined by its URI, and the scheme of the URI SHOULD be used to determine which mechanism shall be used to transfer the resource. Since this is defined in the application then the application SHALL implement instances of `ResourceTransferDelegate` to provide the transfer mechanism. This interface constructs instances of `ResourceReader` and `ResourceWriter` which read and write a resource respectively. On construction, these objects SHALL have opened the resource in a way appropriate to the implementation. The method `ResourceReader::read()` SHALL read bytes into the provided byte array. It SHALL NOT read more bytes into the array than the array can accomodate and shall return the number of bytes written to the array. The return value SHALL be, then, $number\ of\ bytes\ read \leq |buffer|$. Whereas, the method `ResourceWriter::write()` SHALL write the number of bytes in the provided byte array to the implementation defined resource destination. Both `ResourceReader::close()` and `ResourceWriter::close()` SHALL clean up any implementation specific allocations that have been used to read or write the resource in order that the resource is closed.

The method `ResourceTransferDelegate::getResource()` constructs the `Resource` object that will represent the uploaded resource or the resource to be downloaded. This resource SHALL be provided to the application when the resource upload or download observer methods are called. These observers SHOULD be used to implement any application specific operations that shall be processed once a resource has been up- or downloaded, e.g., persist the resource description to a backing store.

It is application defined as to what happens to an anonymous session owned resources once the session is ended. In some applications the resources MAY need to the removed. However, in others perhaps they persist for others to use.

Sessions that require user identity are provided by the classes `UserSession` and `AbstractUserSession`. These classes mixin the ability to store and retrieve the user associated with the session.

Other optional session functionality can be mixed in with the following interfaces:

- `EngineAssignableSession`: used to mixin grant and revoke machine translation engine access to users, and
- `RoleAssignableSession`: used to grant and revoke roles to and from users.

The mixin interfaces `EngineRetrievableSession` and `UserRetrievableSession` allow for the retrieval of engine objects and user objects from some backing store. The retrieval methods are passed a predicate argument that SHALL be used to filter the engine and user objects that are returned. This could be used to formulate an SQL statement that queries a relational backing store to retrieve a session's engines. The user retrieval method, `UserRetrievableSession::retrieveUsers()`, is defined to allow administrators assign engines or roles to users. However, the implementation SHALL have the last word as to exactly what happens in a particular application.

The grant and revoking of MT engine access rights and roles to and from users will be discussed in Section 4.3.

4.2 Session Negotiation

On session creation the server and the client MAY come to some agreement on what a client can expect from a server and what the server can expect from a client. For example, the client may wish to submit a PDF document for translation. If the server cannot handle PDF documents then the client may not wish to start a session since the server is lacking support for the client's future request(s). Conversely, a server that only supports translation memories in XLIFF format is not compatible with a client that only supports TMX. The final choice as to whether a session is finalised for use SHALL rest with the server. However, clients MAY terminate a session at session negotiation time if the server does not support the capabilities the client requires.

Session negotiation is useful when two MT systems need to communicate with each other. The session negotiation API allows the two systems to discover whether they are compatible or provide the services they require. In an MT system where translation jobs are farmed out to other, possibly unknown, systems this discovery is essential since it provides dynamic capability information and allows *ad hoc* system choice: let's give this new service a try since it supports the required capabilities. Session negotiation keeps the knowledge of which capabilities a service provides in the service, not the consumer of the service.

Session negotiation is an optional part of this API. If session negotiation is implemented then negotiation SHALL be completed before the session is fully established and before any client request is serviced. It is application defined as to what course of action to take when capabilities do not match: either tear down the session forcefully or leave the client to decide. However, there may be situations where session negotiation is not required. If the server- and client-side of the API's implementation have static expectations and requirements then session negotiation may not be required since both "sides" of the application intrinsically "know" each other.

Session negotiation SHALL proceed by exchanging server- and client-side *capabilities*.

4.2.1 Capabilities

The capabilities come in four board flavours:

- API: These capabilities determine the nature of the API, i.e., version.
- Resources: these capabilities describe the file types that the service can support. Supporting means that the service will store and use the resource in an appropriate way.
- Features: the operations that can be expected from an MT service but may not be available in every MT service.
- Prerequisites: the prerequisites that the client SHALL ensure are true before some or all of the MT service's features become unavailable to a client, e.g., payment.

Name	Meaning	Group
`API_VERSION`	The semantic version of the API.	API
`RES_FILE_TMX`	*Translation Memory eXchange*[2] (TMX) file format supported.	Resource
`RES_FILE_TBX`	*TermBase eXchange*[4] (TBX) file format supported.	Resource
`RES_FILE_UTX`	*Universal Terminology eXchange*[6] (UTX) file format supported.	Resource
`RES_FILE_SRX`	*Segmentation Rules eXchange*[1] (SRX) file format supported.	Resource
`RES_FILE_ITS`	*Internationalization Tag Set*[3] (ITS) file format supported.	Resource
`RES_FILE_XLIFF`	*XML Localisation Interchange File Format*[5] (XLIFF) file format supported.	Resource
`RES_FILE_TTX`	*SDL Trados translation memory* file format support.	Resource
`FET_RES_UPLOAD`	Sessions can upload resources to the MT server. This feature MAY be disabled for particular sessions or resource types and is application defined.	Feature
`FET_RES_DOWNLOAD`	Session can download resources that are owned by them from the MT service. This feature MAY be disable for particular sessions or resource types and is application defined.	Feature
`PRE_REQ_PAYMENT`	A payment model is in operation for the MT service. The service, at least, requires payment for some operations. Freemium only services SHALL NOT specify this capability.	Prerequisite

Table 4.1: Capability names.

Capabilities are flat representations of aspects of the supported resource kinds, features and expectations. Depending on the implementation these could be strings or enumerations, either way they SHALL be unique and follow the standard naming shown in Table 4.1.

The Trados translation memory format is a proprietory format developed by SDL. It is included since some MT systems may wrap SDL products which can support this translation memory format.

The capability class diagram is shown in Figure 4.2. The unique identifier for a capability is its name as shown in `Capability` interface. The four flavours of capability are shown: the two of note are

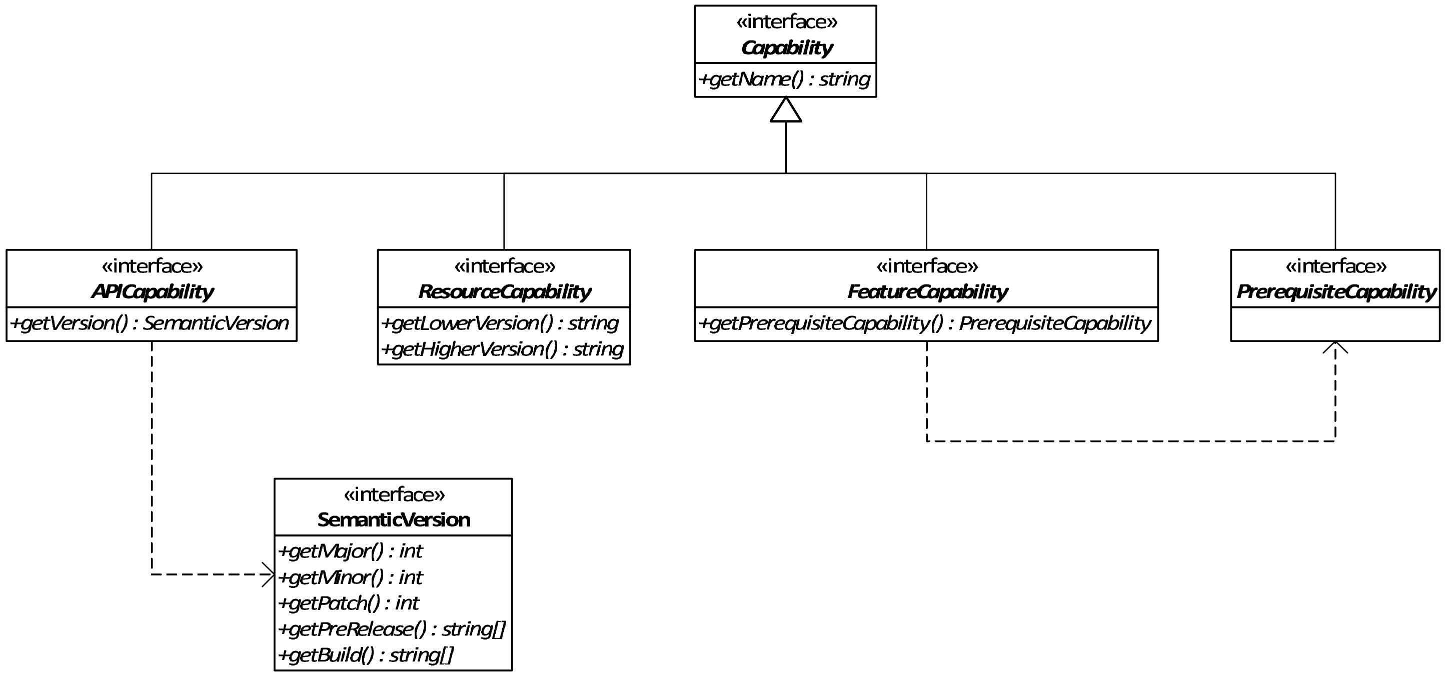

Figure 4.2: Capabilities interface diagram.

`APICapability` and `ResourceCapability`. In order to characterise support for resource kinds the version range is specified by the lowest and highest version supported: see `ResourceCapability::getLowerVersion()` and `ResourceCapability::getHigherVersion()` respectively. These methods return strings which SHALL be meaningful to the implementation. The comparison of these version strings SHALL be the implementation's responsibility.

The API is versioned using *semantic versioning*[1]. Semantic versioning imposes rules on the meaning on the increments of version in order to have an appropriate level of control of dependency versioning. This allows consumers of this API to upgrade easily and safely. A semantic version has the form *X.Y.Z*: which are non-negative integers that denotes the major, minor and patch version of the API. Pre-release and build identifiers can also be added, e.g., 1.5.7-alpha.2+build.2012.01.22. Semantic versioning imposes an ordering so that recent and previous releases can be compared, i.e., 1.0.0-alpha <1.0.0-beta <1.0.0-rc.1 <1.0.0-rc.1+build.1 <1.0.0 <1.3.7+build.1 <2.0.0.

Once the client, after an optional authentication step, has sent its capabilities the MT service SHALL compare them with its capabilities. The MT service SHALL then return the unsupported client capabilities back to the client. If the application determines that a meaningful conversation cannot be carried out with the client then the application server-side SHOULD terminate the session. Otherwise, the decision to terminate the session is delegated to the client. However, this API does not impose *any* restrictions on which party terminates the session, it is completely the responsibility of the implementer to choose the appropriate course of action for the specific application.

More formally, the negotiation class diagram is shown in Figure 4.3. Two interfaces are used to represent a client's capability negotiation request and the response that the application constructs. The response is to be sent back to the client. The client can then decide whether to continue the session if it has not already been torn down by the service. Using the `ClientCapabilityRequest` interface the client specifies:

- The version of the MT API it expects to be presented with (see `ClientCapabilityRequest::getVersionCapability()`), and
- The resource kinds that the client will be expected to be supported (see `ClientCapabilityRequest::getResourceCapabilities()`).

[1]See Semantic Versioning 2.0.0-rc.1 at `http://semver.org/`

Using the `ConcreteNegotiator::negotiate()` the service SHALL construct an instance of the `ClientCapabilityResponse` interface. This interface specifies:

- Whether the client's API version is supported (see `ClientCapabilityResponse::isClientAPISupported()`),
- Which of the resource capabilities, provided by the client, are *not* supported (see `ClientCapabilityResponse::getUnsupportedResourceCapabilities()`), and
- A description of the feature capabilities of the MT service along with their prerequisite capabilities, e.g., whether payment may be needed to use the feature (see `ClientCapabilityResponse::getFeatureCapabilities()`).

Once the negotiation is complete, the service can inspect the response and tear down the client's session if it deems necassary. However, the negotiation response MUST be returned to the client.

4.3 Authorisation

The MT API does not, by itself, support any security features. This is left to the application programmer to integrate with authentication systems if authentication is necessary. However, in order to support operations that do require a user then some integration is needed with an external authentication provider. This interlocks the external authentication and authorisation and the API representation of users and roles present in the application. Two interfaces are used to represent a user, and their authorisation role, in the API, namely `User` and `Role` (see Figure 4.4).

These mixin interfaces SHOULD be used with the application's representation of users and roles to implement the interfaces that require these objects. The method `User::getIdentifier()` SHALL return the user's unique identifier, and `User::getRoles()` SHALL return an array representing the roles assigned to the user. The role interface requires that the method `Role::getCode()` SHALL return a unique code number for the role and `Role::getName()` SHALL return a human readable name for the role. The implementation is free to define its own roles that are appropriate to the authorisation model used. For example, an application may have three roles: superuser, administrator and normal user. These roles will have different authorisations administrative, construction and translation operations. The `Role` interface can be used to integrate an implementation's roles into the API implementation (see `UserRetrievableSession`, `EngineAssignableSession`, and `RoleAssignableSession` interfaces in Figure 4.1).

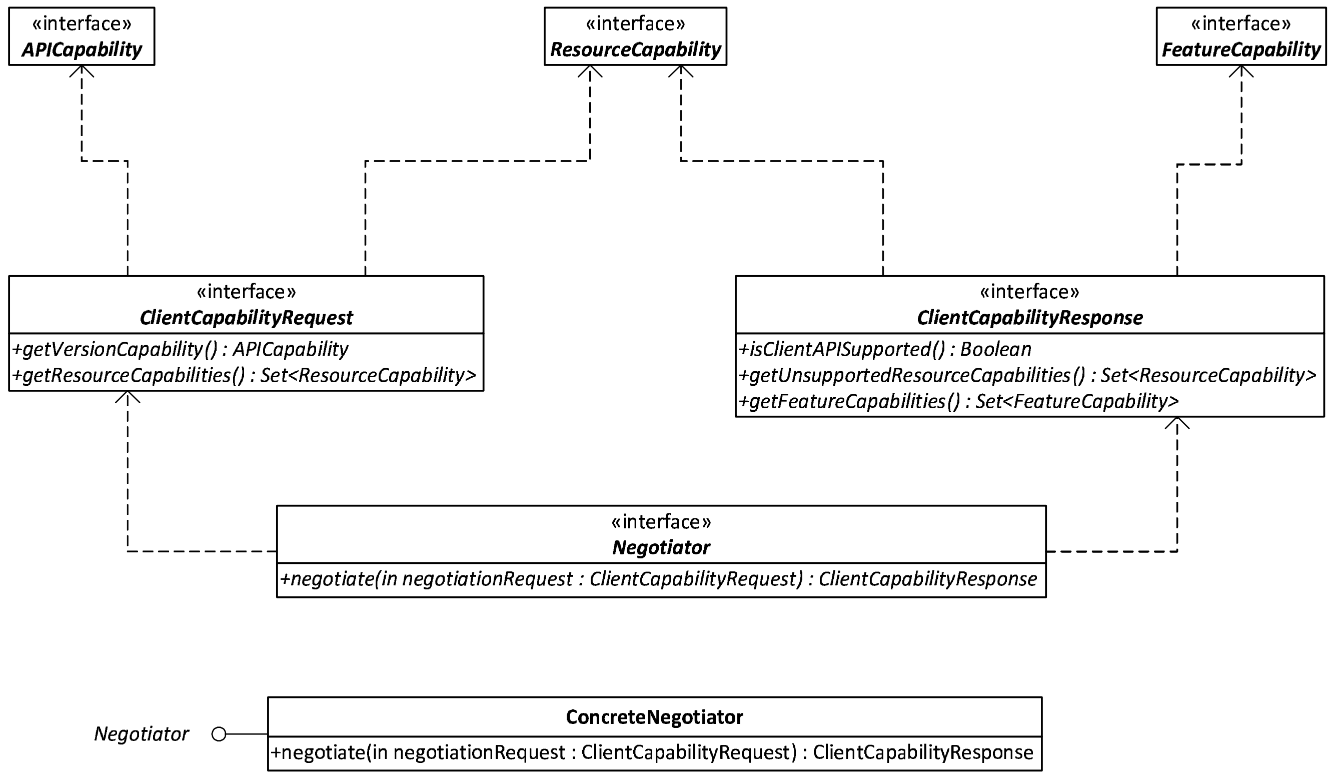
«interface»
APICapability
«interface»
ResourceCapability
«interface»
FeatureCapability
«interface»
ClientCapabilityRequest
+getVersionCapability() : APICapability
+getResourceCapabilities() : Set<ResourceCapability>
«interface»
ClientCapabilityResponse
+isClientAPISupported() : Boolean
+getUnsupportedResourceCapabilities() : Set<ResourceCapability>
+getFeatureCapabilities() : Set<FeatureCapability>
«interface»
Negotiator
+negotiate(in negotiationRequest : ClientCapabilityRequest) : ClientCapabilityResponse
Negotiator
ConcreteNegotiator
+negotiate(in negotiationRequest : ClientCapabilityRequest) : ClientCapabilityResponse

Figure 4.3: Capability negotiation interface diagram.

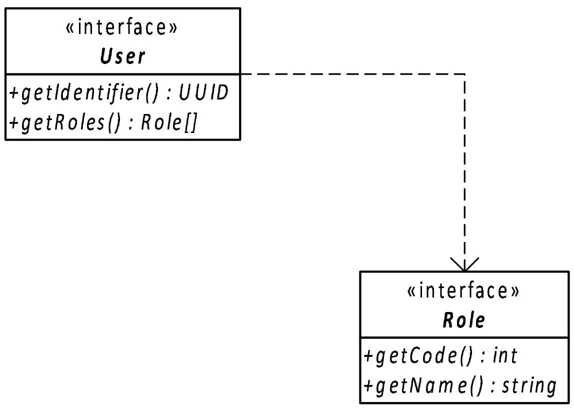

Figure 4.4: Authorisation interface diagram.

Chapter 5

Scheduling

Machine translation consists of a number of operations which are computationally expensive. Constructing an MT service with many users requires that the computational resources are shared *fairly* between the demands of the users. The implementer of an MT service needs to define:

- Which computational resource or resources will be used to execute the computationally expensive operations,
- The latency of an operation before it is executed, and
- A policy to determine how user's operations will be scheduled, i.e., priority.

The scheduling API also needs to support different kinds of computation resource management: from native threading to distributed resource management products. The pattern used in the scheduling API is detached execution with notification on completion, whether successful or not. The implementer of the API is able to choose which technology to use for the detached execution: thread-pools, Open Grid Engine, or Amazon EC2 etc.

5.1 Tickets

The scheduling API SHALL issue *tickets* when an operation is submitted to the underlying detached execution implementation. A ticket is a receipt for, and uniquely identifies an operation. When the operation is submitted an *observer* SHALL be provided which observes the progressing computation. On completion, the observer is invoked with the appropriate ticket to identify which operation has completed. This is the observer design pattern. The observer is application defined and the application programmer SHOULD take this opportunity to update any data that relies on the computation.

Operation priorities SHALL be defined using the scheduling API. This allows an application defined priority to be used to prioritise operations into the particular detached execution environment. For example, a priority could, say, for a paid-for MT service prioritise operations, invoked by users, which are on a higher tariff. So, say, a user on a *Freemium* tariff would have their operations prioritised lower than a user who pays for the service. Depending on the detached execution environment a priority MAY determine, not only, the latency of an operation, but also how much processor time a certain operation can expect when being execute, c.f., Unix nice levels.

Figure 5.1 shows the scheduling ticket class hierarchy. The ticket classes are structures which only hold data pertinent about the submitted operation. Instances of these classes SHALL be constructed by the methods, found in the API, for machine translation engines and translators (see Section 6.2 and Chapter 7 for method that issue tickets for machine translation engines and translator objects).

The scheduling API provides ticket types for the operations available with MT engines (see Chapter 6), and tranlsation tasks (see Chapter 7). Tickets are derived from an abstract class called `Ticket` and SHALL be uniquely identified by a UUID. They also hold the following data:

- *Start date*: A date/time stamp which is the date at which the ticket was issued. This SHALL be the submission date of the task associated with the ticket.

- *Session*: The session that submitted the task.

- *Priority*: The priority value that was used to schedule the ticketed task.

- *End date*: A *protected* date/time stamp which denotes when the task completed, successfully or not.

Sub-classes of `Ticket` specialise to computationally expensive MT engine operations and translations: the classes `EngineTicket` and `TranslationTicket` respectively. Instances of `EngineTicket` are associated with an MT engine. This is the engine on which the operation is being done and is labeled as the `participatingEngine` in Figure 5.1. The ticket for composing MT engines is associated with two MT engine instances. The engine with which the participating engine is being composed, and the MT engine that will be created due to the

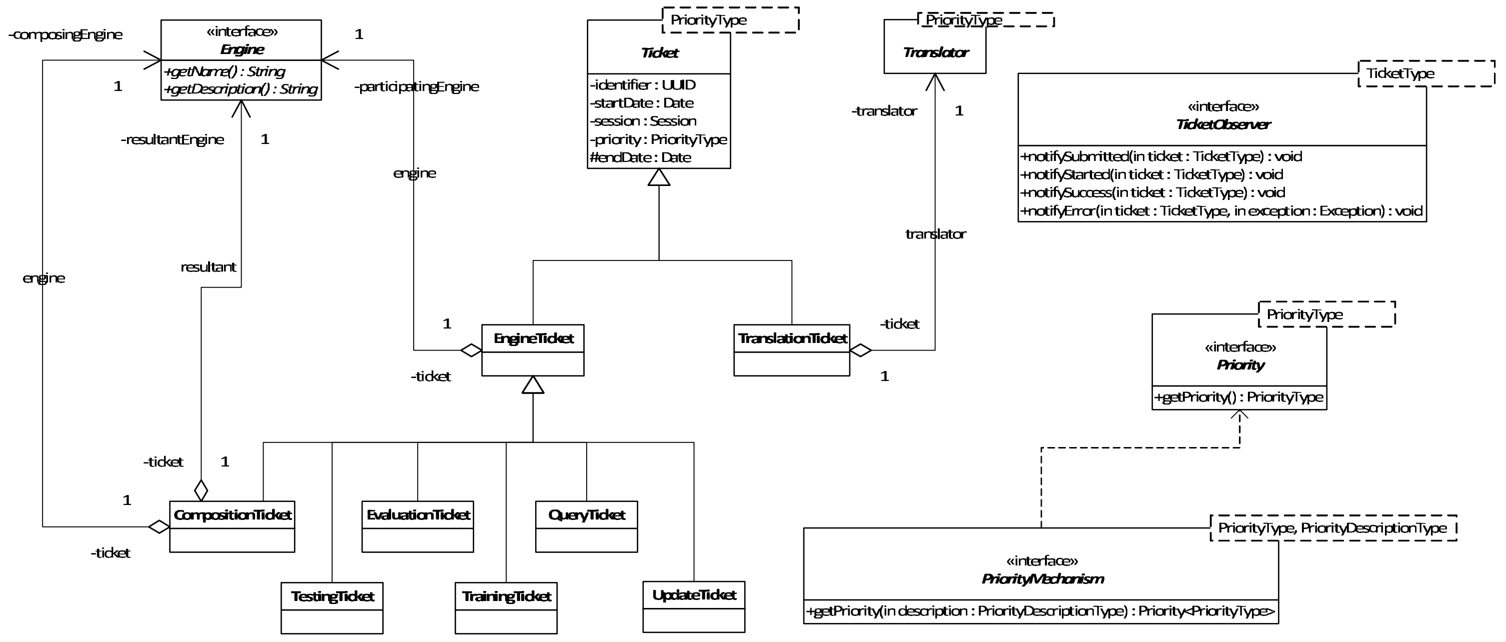

Figure 5.1: Scheduling ticket class diagram.

composition, i.e., the resultant MT engine. Translation tickets are associated with a translator object and will be described in Chapter 7.

Ticket observers are used to notify the world of changes in ticket state. The `TicketObserver` interface defines the events which application programmers can use to update application state. All of the `TicketObserver` methods SHALL be passed the ticket for which the task is associated. The interface defines the following events:

- *Submission*: The implementation is called back once the ticketed task has been submitted to the detached execution environment. See `TicketObserver::notifySubmitted()`.
- *Start*: When the task has been started in the detached execution environment this event is raised. See `TicketObserver::notifyStarted()`.
- *Successful completion*: On successful completion of the ticket task this event is raised. See `TicketObserver::notifySuccess()`.
- *Error*: If at any point an error occurs with the task of execution environment the error callback is executed. The exception that is generated from the error is provided with the event. See `TicketObserver::notifyError()`.

Implementers of the `TicketObserver` MAY implement all, some or none of the methods. It is application defined as to which ticket event are of interest.

5.2 Priority

Two generic interfaces, shown in Figure 5.1, define the prioritising features of the scheduling API. The `PriorityMechanism` is a factory interface that constructs objects of type `Priority`. The mechanism takes an application defined priority description and contructs an object that SHALL provide the priority for a submitted operation. The `PriorityMechanism::getPriority()` method SHALL be used to construct the priority object and MAY be nondeterministic. The priority description SHOULD contain data that can be used to produce a priority that is appropriate for the application. For example, a user's payment plan and remaining credit could determine how their ticketed tasks are scheduled in a paid-for MT service. Maybe the time of day is important in producing a priority, e.g., tasks scheduled at night may attract a higher priority for some users.

Once the `Priority` object has been constructed the actual priority value SHALL be retrieved by calling the `Priority::getPriority()` method. This method SHALL be idempotent. The priority value is stored in the ticket object and SHALL be used to schedule tasks into the detached execution environment.

Chapter 6

Machine Translation Engines

Machine translation engines are the entities that produce translations for, possibly unseen, sentences. Engines are built using primary resources and generally use computationally expensive operations to produce translations. Engines shall have operations available that, depending on their nature, shall read or mutate engine state, e.g.

- Evaluating a engine,
- Composing engines,
- Testing engines, and
- Training SMT engines.

6.1 Engines

An engine is defined as the entity that will perform machine translation. This may be a decoding pipeline in an SMT system or software that implements a rule based system. Engines are modelled as a derived resource and Figure 6.1 shows it's relationship to the resource interfaces.

The application programmer SHALL implement and extend this interface hierarchy to suit the representation for the particular flavour of MT engine being employed. The interface `Engine` allows the programmer to define a name and some descriptive text for an MT engine: `Engine::getName()` and `Engine::getDescription()` respectively. The creation resources, defined using the `DerivedResource::getCreationResources()`, SHOULD reference

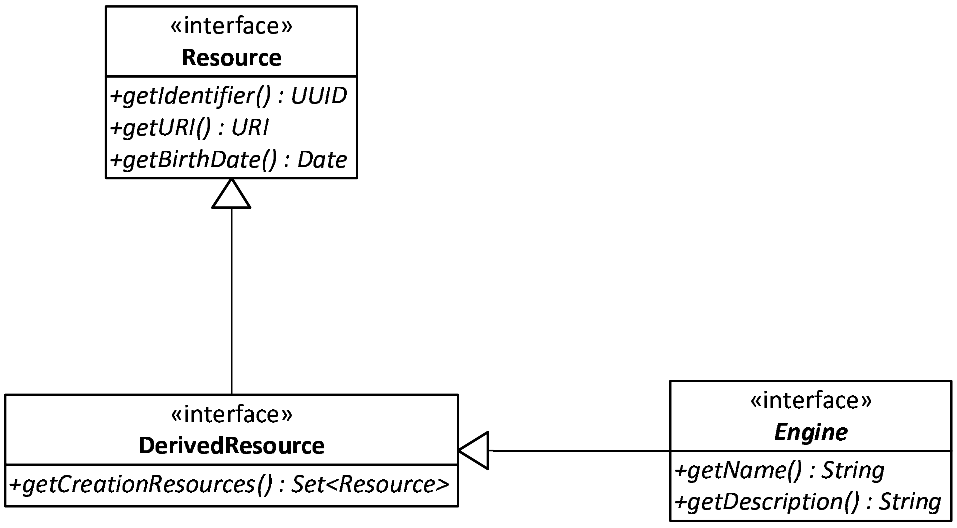

Figure 6.1: MT Engine class diagram.

the primary resources used to construct an engine. This allows the provenance of the engine to be preserved in the representation.

6.2 Optional Engine Functionality

Mixin interfaces are used to add optional functionality to an MT engine. This allows the application programmer to choose mixins useful to the kind of MT engine being constructed. Using mixins in this way prevents the application programmer from being tied to this API; it does not mandate that any class inheritance is used. This is particularly useful when using languages that do not support multiple inheritance and can be used alongside existing frameworks and class hierarchies.

The mixins provided define the following operations:

- Composition: compose one MT engine with another,
- Evaluation: score an MT engine,
- Update parameters: mutate runtime options/parameters,
- Querying: invoking translations one sentence at a time,
- Training and retraining: for SMT engines to build appropriate models,
- Testing: provide resources to test a constructed MT engine, and

- Updating: mutation of an existing engine to adapt to new data or rules.

The operations, in the mixins, that could represent computationally expensive operations use a asynchronous invocation pattern. In order to track the operation the caller of these methods receives a *ticket*. The ticket is used to represent an “in flight” operation and, once complete, SHALL be used in a notification. Notifications are used to inform the application of the state of a completed operation: submitted, starting, or completed successfully or failed.

Figure 6.2 shows the mixin interfaces for optional engine functionality. All of the mixin interface methods MUST be passed a session object. This session object SHALL represent the session that is invoking the operation. Sessions are described in Section 4.1.

6.2.1 Ticketable Engines

The `TicketableEngine` mixin defines a method that SHALL be used to retrieve the issued tickets for operations on an engine. The `TicketableEngine::retrieveTickets()` takes a predicate object that SHALL be used to filter the tickets and return a set of `EngineTicket` objects which meets the predicate. It is implementation defined as to which engine tickets are returned when user authorisation is employed. Only users that are authorised to retrieve some or all engine tickets SHOULD be taken into account when implementing `TicketableEngine` interface. This is the base interface of all the engine mixin interfaces.

6.2.2 Composing Engines

The composition mixin allows applications to chain engines together. This is equivalent to function composition, i.e., suppose two engines E_1 and E_2. Executing the `ComposableEngine::compose()` method using E_1 as the owning object, i.e., $E_1.compose(E_2)$, then the resultant engine is:

$$E_{resultant} = E_2(E_1(s)),$$

where s is a sentence begin translated.

The data exchange format for the engines is also defined in the formal parameters of the compose method. The interface `DataExchangeFormat` specifies which format is to be used to exchange data from engine, E_1, to engine, E_2. The exchange format to use could be gleaned from session negotiation (see more on session negotiation in Section 4.2).

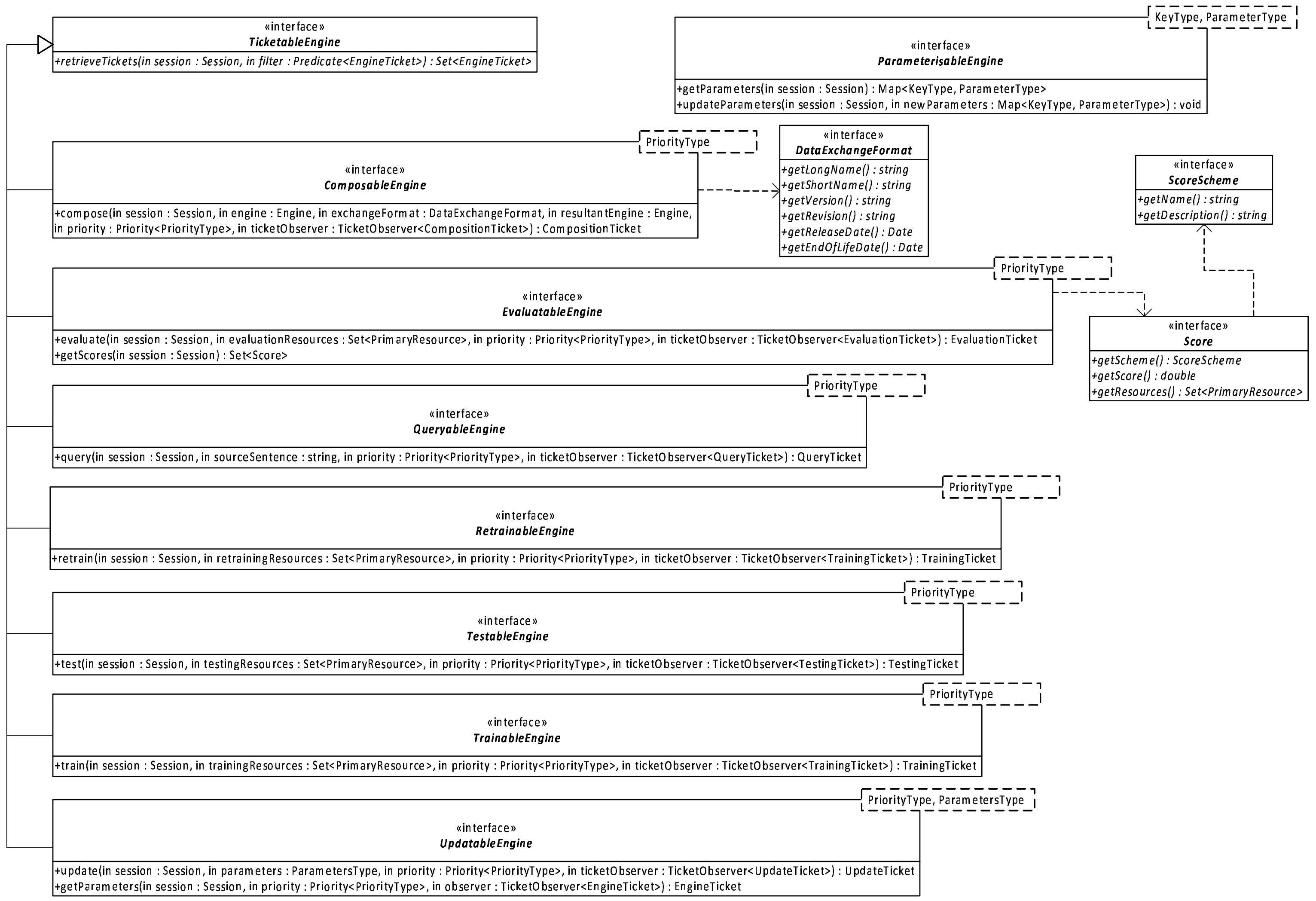
«interface»
TicketableEngine
+retrieveTickets(in session : Session, in filter : Predicate<EngineTicket>) : Set<EngineTicket>
KeyType, ParameterType
«interface»
ParameterisableEngine
+getParameters(in session : Session) : Map<KeyType, ParameterType>
+updateParameters(in session : Session, in newParameters : Map<KeyType, ParameterType>) : void
PriorityType
«interface»
ComposableEngine
+compose(in session : Session, in engine : Engine, in exchangeFormat : DataExchangeFormat, in resultantEngine : Engine, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<CompositionTicket>) : CompositionTicket
«interface»
DataExchangeFormat
+getLongName() : string
+getShortName() : string
+getVersion() : string
+getRevision() : string
+getReleaseDate() : Date
+getEndOfLifeDate() : Date
«interface»
ScoreScheme
+getName() : string
+getDescription() : string
PriorityType
«interface»
EvaluatableEngine
+evaluate(in session : Session, in evaluationResources : Set<PrimaryResource>, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<EvaluationTicket>) : EvaluationTicket
+getScores(in session : Session) : Set<Score>
«interface»
Score
+getScheme() : ScoreScheme
+getScore() : double
+getResources() : Set<PrimaryResource>
PriorityType
«interface»
QueryableEngine
+query(in session : Session, in sourceSentence : string, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<QueryTicket>) : QueryTicket
PriorityType
«interface»
RetrainableEngine
+retrain(in session : Session, in retrainingResources : Set<PrimaryResource>, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<TrainingTicket>) : TrainingTicket
PriorityType
«interface»
TestableEngine
+test(in session : Session, in testingResources : Set<PrimaryResource>, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<TestingTicket>) : TestingTicket
PriorityType
«interface»
TrainableEngine
+train(in session : Session, in trainingResources : Set<PrimaryResource>, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<TrainingTicket>) : TrainingTicket
PriorityType, ParametersType
«interface»
UpdatableEngine
+update(in session : Session, in parameters : ParametersType, in priority : Priority<PriorityType>, in ticketObserver : TicketObserver<UpdateTicket>) : UpdateTicket
+getParameters(in session : Session, in priority : Priority<PriorityType>, in observer : TicketObserver<EngineTicket>) : EngineTicket

Figure 6.2: MT Engine mixin class diagram.

Since this operator could take a considerable amount of time this operation is ticketed and prioritised. Therefore, an observer SHALL be provided that shall receive notifications. Also, a `CompositionTicket` is returned to the caller.

6.2.3 Evaluating Engines

One important benchmark of an engine's translation performance is scoring. Many scoring schemes exist, e.g., BLEU and METEOR, and the `EvaluatableEngine` mixin allows engines to be ranked using any scoring scheme. The `EvaluatableEngine::evaluate()` method SHALL be implemented for the scoring scheme(s) used. The primary resources used to generate the score are passed into the method and a `EvaluationTicket` is returned. Since this is a ticketed operation a ticket observer is required which will receive notifications as the evalutation operation proceeds. This potentially long running task is also prioritised. Once complete a record of the score or scores SHALL be constructed using the `Score` and `ScoreScheme` interfaces. The `EvalutableEngine::getScores()` method SHALL be used to retrieve the scores for the engine.

The `Score` interface defines:

- *Scoring Scheme*: The `ScoringScheme` defines the scheme used and SHALL be retrieved with the `Score::getScheme()` method.
- *Score*: A double precision value which is the generated score. The `Score::getScore()` method SHALL be used to retrieve the score.
- *Resource Provanence*: Any primary resources that were used to generate the score SHALL be retrievable using the `Score::getResources()`.

6.2.4 Queryable Engines

Users of MT systems require that translations are done on a sentence-by-sentence basis to test a constructed engine. The `QueryableEngine` mixin supports this use case. Invoking the `QueryableEngine::query()` method with a sentence, and a priority SHALL return query ticket. This operation is ticketed since the it could take a considerable amount of time, even though it is only a single sentence. An observer SHALL be provided to notify the application of the query operation.

6.2.5 Trainable and Re-trainable Engines

The `TrainableEngine` and `RetrainableEngine` mixin interfaces are included for SMT systems. However, it is not unreasonable that these mixins will be used in a non-SMT engine implementation. Typically training and re-training an SMT engine can take a considerable amount of time even using small corpora. All operations in these mixin interfaces are ticketed and prioritised. As with all the ticketed operations, a ticket observer SHALL be required. Primary resources are used to train and re-train SMT engine and a collection of these instances SHALL be passed to the `TrainableEngine::train()` and `RetrainableEngine::retrain()` methods.

6.2.6 Testable Engines

Testing a constructed or updated engine is a crucial step in developing machine translation engines: the developer needs some level on confidence that the translation performance of an engine meets some benchmark or criteria. Testing an engine could take a considerable amount of time and, therefore, the `TestableEngine` mixin interface tickets the testing operation. The `Testable::test()` method SHALL be implemented to carry out ticketed testing tasks. This method SHALL be given a ticket observer, a priority, and a collection of primary resources to use as test vectors.

6.2.7 Updatable Engines

Engines from time-to-time will require maintenance. In rule-based systems new rules could be required, and in SMT based systems tuning operations may be needed once training is complete. The `UpdatableEngine` mixin allows for these potentially long running and mutating operations to be implemented. The engine's updatable parameters SHALL be fetched by implementing the `UpdatableEngine::getParameters()` method. Since retrieving parameters could potentially take some time it is ticketed and prioritised.

Updating, or mutating, the engine SHALL be achieved by implementing the `UpdatableEngine::update()` method. This operation is ticketed and requires a ticket observer. A priority is used to fairly schedule this updating request.

6.2.8 Parameterisable Engines

Machine translation engine software MAY require options and settings in order to configure the operation of an engine. These name-value pairs are called *parameters* in this API. The `ParameterisableEngine` interface SHALL mixin a getter and a setter for engine parameters. To retrieve an engine's current parameters the `ParameterisableEngine::getParameters()` SHALL be used. Commiting parameters back to the engine SHALL be done using the `ParameterisableEngine::updateParameters()` method. These operations are expected to complete quickly, since they are accessing files or memory, and are therefore *not* ticketed.

Chapter 7

Translators

Translators are a conglomeration of an MT engine, translation memories and glossaries. A translator SHALL specify at least one of these resources. This allows translators to support translations using any combination of MT, TM or glossaries. It is the responsibility of application programmers to handle these resources in an appropriate way for the flavour of translation required.

Translations are typically computationally expensive and can take a considerable amount of time to complete. In an MT system that is multi-user computation resources should be shared fairly between the demands of the submitted translations. As with MT engine operations (see Chapter 6), translations SHALL be ticketed and a ticket observer is REQUIRED to receive notifications of the progress of a translation task.

The standard models a translator as a derived resource (see `DerivedResource` interface in Figure 3.1). An abstract and associative class called `Translator` is defined, and is shown in Figure 7.1. There are two methods for performing a translation:

- *Primary Resource Translation*: A primary resource uploaded to the MT system can be translated. It is application defined as to which kind of primary resources are supported for translation. If supported, it is implementation defined as to whether any pre- or port-processing is required, e.g. file filtering. If the primary resource is not supported the *TicketObserver::notifyError()* SHOULD be called immediately. For a description of ticket observers see Section 5.1.

- *Sentence-by-sentence Translation*: Translations can be supported that consist of a single sentence. Recall that MT engines can be queried sentence-by-

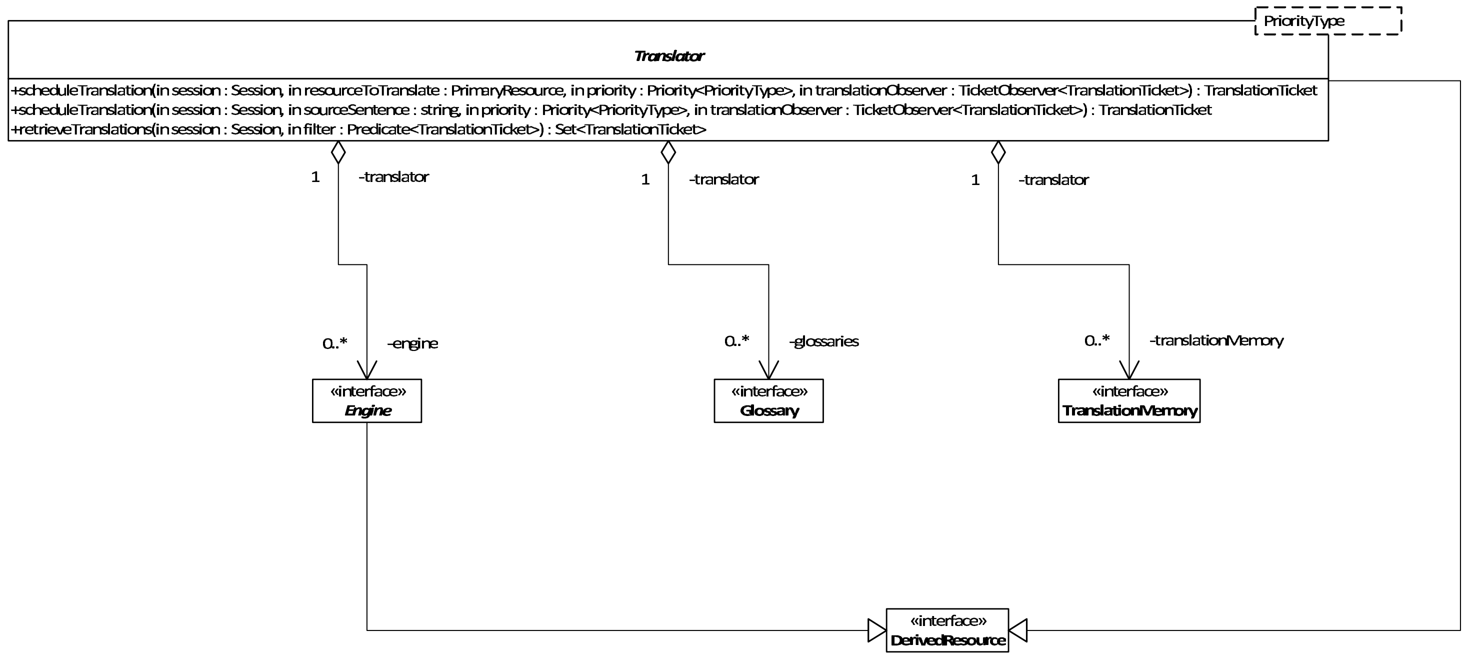
PriorityType
Translator
+scheduleTranslation(in session : Session, in resourceToTranslate : PrimaryResource, in priority : Priority<PriorityType>, in translationObserver : TicketObserver<TranslationTicket>) : TranslationTicket
+scheduleTranslation(in session : Session, in sourceSentence : string, in priority : Priority<PriorityType>, in translationObserver : TicketObserver<TranslationTicket>) : TranslationTicket
+retrieveTranslations(in session : Session, in filter : Predicate<TranslationTicket>) : Set<TranslationTicket>
1
-translator
1
-translator
1
-translator
0..*
-engine
0..*
-glossaries
0..*
-translationMemory
«interface»
Engine
«interface»
Glossary
«interface»
TranslationMemory
«interface»
DerivedResource

Figure 7.1: Translator class diagram.

sentence (see Section 6.2.4) to perform a translation using only the engine. However here, TM and glossaries can be mixed into a richer translation that uses any translation pipeline that may be implemented.

The methods defined in the `Translator` abstract class SHALL be passed a session object. This session object SHALL represent the session that is invoking the operation. A detailed description of sessions can be found in Section 4.1.

The abstract class `Translator` holds references to, optionally, an MT engine, translation memories and glossaries. Translations are scheduled using the polymorphic method `Translator::scheduleTranslation()`. One version of this method takes a primary resource object, and the other a string object, these methods SHOULD be used for document and sentence-by-sentence translation respectively. The polymorphic translation method SHALL be provided with a priority and a ticket observer and returns a `TranslationTicket` object as a receipt for the translation task. This ticket SHALL be made available in the ticket observer callback methods.

A single translator can be used to schedule many translations, i.e., the same resources used to translate many primary resources or sentences. To query the translations that have been scheduled but not completed the `Translation::retrieveTranslations()` method SHALL be implemented. This method takes a predicate object to filter the translations. It is implementation defined as to which user's sessions are authorised to retrieve which translation tickets. The authorisation model SHALL be application defined but MAY be dependent on which user/session owns a translator object.

Implementers of the abstract `Translator` class should extend it to use their favourite detached execution environment to execute the translation tasks. It is left to the application programmer to decide how the computational resources are shared between translations. For example, using a distributed resource management system such a cluster or grid computing separate job queues can be configured to segregate the logically disparate translation tasks.

Bibliography

[1] SRX Standard. http://okapi.opentag.com/help/lib/ui/segmentation/srx.html.

[2] TMX 1.4b Specification. http://www.gala-global.org/oscarStandards/tmx/, 2005.

[3] Internationalization Tag Set (ITS) Version 1.0. http://www.w3.org/TR/its/, 2007.

[4] Systems to manage terminology, knowledge, and content - TermBase eXchange (TBX). http://www.ttt.org/oscarStandards/tbx/, 2008.

[5] XLIFF Version 1.2. http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html, 2008.

[6] UTX Specification Version 1.11. http://www.aamt.info/english/utx/utx1.11-specification- 2011.