JADE TUTORIAL

APPLICATION-DEFINED CONTENT LANGUAGES AND ONTOLOGIES

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 30-June-2002. JADE 2.6

Authors: Giovanni Caire (TILAB, formerly CSELT)

TABLE OF CONTENTS

APPLICATION-DEFINED CONTENT LANGUAGES AND ONTOLOGIES

This section describes the new content languages and ontologies support provided by JADE since version 2.5 and included in the `jade.content` package. Former versions of JADE already included an "old" support for content languages and ontologies (scattered around the `jade.onto`, `jade.lang.sl` and `jade.core` packages) that is still available for backward compatibility. With respect to the old one, this new support provides a number of advanced features, such as the possibility of creating queries and using content language operators, that can be extremely important in complex applications.

As usual in JADE, the approach is "pay as you go": using the basic features (covered in 4), that are equivalent to the old support, is as simple as (and very similar to) using the old support. Using the advanced features (covered in 5) requires the developer to know more about the internal operations of this new content languages and ontologies support. Finally section 7 describes how to modify an application written for the old support in order to make it use the new one.

1 RATIONALE

When an agent A communicates with another agent B, a certain amount of information *I* is transferred from A to B by means of an ACL message. Inside the ACL message, *I* is represented as a content expression consistent with a proper content language (e.g. SL) and encoded in a proper format (e.g. string). Both A and B have their own (possibly different) way of internally representing *I*. Taking into account that the way an agent internally represents a piece information must allow an easy handling of that piece of information, it is quite clear that the representation used in an ACL content expression is not suitable for the inside of an agent.

For example the information that *there is a person whose name is Giovanni and who is 33 years old* in an ACL content expression could be represented as the string

```
(Person :name Giovanni :age 33)
```

Storing this information inside an agent simply as a string variable is not suitable to handle the information as e.g. getting the age of Giovanni would require each time to parse the string.

Considering software agents written in Java (as JADE agents are), information can conveniently be represented inside an agent as Java objects. For example representing the above information about Giovanni as an instance (a Java object) of an application-specific class

```
class Person {
      String name;
      int age;

      public String getName() {return name; }
      public void setName(String n) {name = n; }
      public int getAge() {return age; }
      public void setAge(int a) {age = a; }
      ….
}
```

initialized with

```
name = "Giovanni";
age = 33;
```

would allow to handle it very easily.

It is clear however that, if on the one hand information handling inside an agent is eased, on the other hand each time agent A sends a piece of information *I* to agent B,

1) A needs to convert his internal representation of *I* into the corresponding ACL content expression representation and B needs to perform the opposite conversion.
2) Moreover B should also perform a number of semantic checks to verify that *I* is a meaningful[1] piece of information, i.e. that it complies with the rules (for instance that the age of Giovanni is actually an integer value) of the ontology by means of which both A and B ascribe a proper meaning to *I*.

The support for content languages and ontologies provided by JADE is designed to automatically perform all the above conversion and check operations as depicted in Figure 1, thus allowing developers manipulating information within their agents as Java objects (as described above) without the need of any extra work.

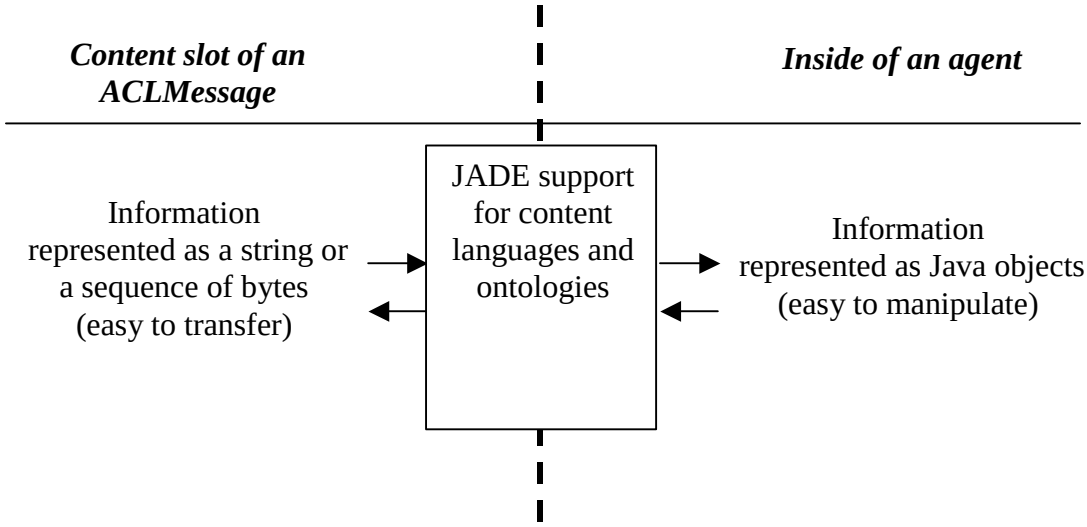

Figure 1The conversion performed by the JADE support for content languages and ontologies

2 MAIN ELEMENTS

The conversion and check operations described in 1 are carried out by a **content manager** object (i.e. an instance of the `ContentManager` class included in the `jade.content` package). Each JADE agent embeds a content manager accessible through the `getContentManager()` method of the `Agent` class. The `ContentManager` class provides all the methods to transform Java objects into strings (or sequences of bytes) and to insert them in the content slot of `ACLMessages` and vice-versa.

The content manager provides a convenient interfaces to access the conversion functionality, but actually just delegates the conversion and check operations to an **ontology** (i.e. an instance of the `Ontology` class included in the `jade.content.onto` package) and a **content language codec** (i.e. an instance of the `Codec` interface included in the `jade.content.lang` package). More specifically the ontology validates the information to be converted from the semantic point of view

[1] Not necessarily true. If I say that "the age of Giovanni is 34", this information is meaningful (a meaningless information would be for example that "the age of Giovanni is dog"), but can be false as maybe Giovanni is 33.

while the codec performs the translation into strings (or sequences of bytes) according to the syntactic rules of the related content language. These operations are described in more details in 5.3, but the user does not need to care about them unless he needs to use some advanced features such as performing queries.

3 THE CONTENT REFERENCE MODEL

In order for JADE to perform the proper semantic checks on a given content expression it is necessary to classify all possible elements in the domain of discourse (i.e. elements that can appear within a valid sentence sent by an agent as the content of an ACL message) according to their generic semantic characteristics. This classification is derived from the ACL language defined in FIPA that requires the content of each ACLMessage to have a proper semantics according to the performative of the ACLMessage. More in details at the first level we distinguish between predicates and terms.

Predicates are expressions that say something about the status of the world and can be `true` or `false` e.g.

```
(Works-for (Person :name John) (Company :name TILAB))
```

stating that "the person John works for the company TILAB".

Predicates can be meaningfully used for instance as the content of an INFORM or QUERY-IF message, while would make no sense if used as the content of a REQUEST message.

Terms are expressions identifying entities (abstract or concrete) that "exist" in the world and that agents talk and reason about. They are further classified into:

Concepts i.e. expressions that indicate entities with a complex structure that can be defined in terms of slots e.g.

```
(Person :name John :age 33)
```

Concepts typically make no sense if used directly as the content of an ACL message. I general they are referenced inside predicates and other concepts such as in

```
(Book :title "The Lord of the rings" :author (Person :name "J.R.R. Tolkjien"))
```

Agent actions i.e. special concepts that indicate actions that can be performed by some agents e.g.

```
(Sell (Book :title "The Lord of the rings") (Person :name John))
```

It is useful to treat agent actions separately since, unlike "normal" concepts, they are meaningful contents of certain types of ACLMessage such as REQUEST. Communicative acts (i.e. ACL messages) are themselves agent actions.

Primitives i.e. expressions that indicate atomic entities such as strings and integers.

Aggregates i.e. expressions indicating entities that are groups of other entities e.g.

```
(sequence (Person :name John) (Person :name Bill))
```

Identifying Referential Expressions (IRE) i.e. expressions that identify the entity (or entities) for which a given predicate is true e.g.

```
(all ?x (Works-for ?x (Company :name TILAB))
```

identifying "all the elements x for which the predicate `(Works-for x (Company :name TILAB))` is `true`, i.e. all the persons that works for the company TILAB).

These expressions are typically used in queries (e.g. as the content of a QUERY_REF) message) and requires variables.

Variables i.e. expressions (typically used in queries) that indicate a generic element not known a-priori.

A fully expressive content language should be able to represent and distinguish between all the above types of element. An ontology for a given domain is a set of schemas defining the structure of the predicates, agent actions and concepts (basically their names and their slots) that are pertinent to that domain.

The final Content Reference Model (depicted in Figure 2) includes two more types of element that are introduced considering that only predicates, agent actions, IREs and lists of elements of these three types[2] (**ContentElementList**) are meaningful content of at least one ACL message. They all inherit from the **ContentElement** super-type.

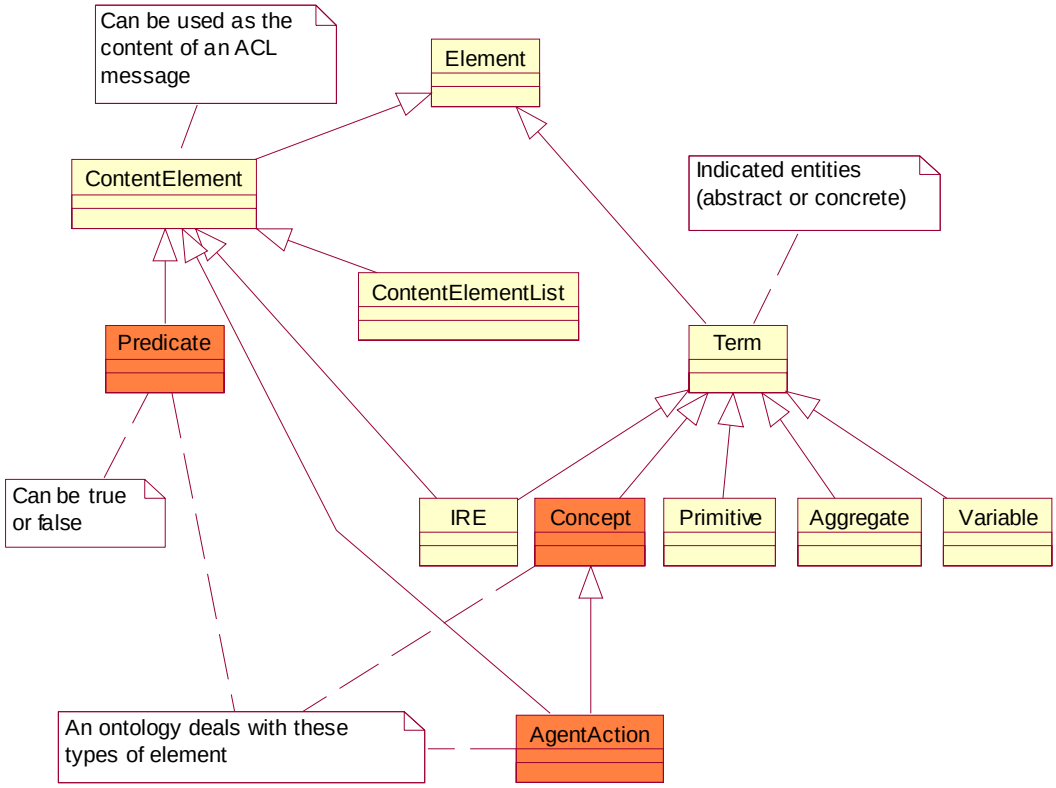

Figure 2The Content Reference Model

4 USING THE JADE CONTENT LANGUAGE AND ONTOLOGY SUPPORT. BASIC FEATURES

This section describes the basic features of the new JADE content language and ontology support included in the `jade.content` package. These features are typically sufficient in a lot of normal cases and, for users already familiar with JADE, they exactly correspond to those provided by the previous JADE support. The `jade.content` package however provides a number of advanced features, such as the possibility of creating queries, that are important for complex applications and that were not available in the old support. These advanced features are covered in 5.

[2] E.g. The content of a PROPOSE message is an agent action + a predicate indicating the conditions that will become `true` if the agent action is performed.

Exploiting the JADE content language and ontology support included in the `jade.content` package to make agents talk and reason about "things and facts" related to a given domain goes through the following steps.

1) Defining an ontology including the schemas for the types of predicate, agent action and concept that are pertinent to the addressed domain. This is discussed in 4.2.
2) Developing proper Java classes for all types of predicate, agent action and concept in the ontology. This is discussed in 4.3.
3) Selecting a suitable content language among those directly supported by JADE. This is discussed in 4.4. JADE can be easily extended to support new user-defined content languages (see 5.8), but in the great majority of the cases the user does not need to define his own content language.
4) Registering the defined ontology and the selected content language to the agent. This is discussed in 4.5.
5) Creating and handling content expression as Java objects that are instances of the classes developed in step 2 and let JADE translate these Java objects to/from strings or sequences of bytes that fit the `content` slot of `ACLMessages`. This is discussed in 4.6.

The above steps are illustrated by means of a simple example described in 4.1.

4.1 The Music Shop example

This example considers a Seller agent managing a simple electronic music shop where two types of item (i.e. CDs and musical books) are available for sale. Each item has a serial number. Each CD has a name and a number of tracks each one has a title and a duration. Each musical book has a title. The Seller agent owns a number of items and can sell them to other Buyer agents.
In the example a Buyer agent will ask the Seller agent if he owns a given CD and, if this is the case, he will request him to sell that CD.

4.2 Defining an Ontology

An ontology in JADE is an instance of the `jade.content.onto.Ontology` class to which the schemas defining the structure of the types of predicates, agent actions and concepts relevant to the addressed domain have been added. These schemas are instances of the `PredicateSchema`, `AgentActionSchema` and `ConceptSchema` classes included in the `jade.content.schema` package. These classes have methods by means of which it is possible to declare the slots that defines the structure of each type of predicate, agent action and concept.
As an ontology is basically a collection of schemas that typically does not evolve during an agent lifetime, it is a good practice to declare the ontology as a singleton object and to define an ad-hoc class (that extends `jade.content.onto.Ontology`) with a static method to access this singleton object. This allows sharing the same ontology object (and all the included schemas) among different agents in the same JVM.
In the music shop example we will deal with four concepts (Item, CD, Track and Book), one predicate (Owns) and one agent action (Sell). Besides them we will have to deal with the concept of AID. The latter however does not need to be defined as each ontology in JADE normally extends a basic ontology (represented as a singleton object of the `jade.content.onto.BasicOntology` class) that includes the schemas for

- the primitive types (`STRING, INTEGER, FLOAT...`)
- the aggregate type

- some generic (i.e. not belonging to any specific domain) predicates, agent actions and concepts among which the AID concept identifying an agent.

In order to declare that the ontology `O1` extends the ontology `O2` (i.e. all predicates, agent actions and concepts included in `O2` are also included in `O1`) it is sufficient to pass `O2` as a parameter when `O1` is constructed.

Taking into account all the above issues, the ontology for the music shop domain can be defined as

```
package musicShopOntology;

import jade.content.onto.*;
import jade.content.schema.*;

public class MusicShopOntology extends Ontology {
  // The name identifying this ontology
  public static final String ONTOLOGY_NAME = "Music-shop-ontology";

  // VOCABULARY
  public static final String ITEM = "Item";
  public static final String ITEM_SERIAL = "serial-number";

  public static final String CD = "CD";
  public static final String CD_NAME = "name";
  public static final String CD_TRACKS = "tracks";

  public static final String TRACK = "Track";
  public static final String TRACK_TITLE = "title";
  public static final String TRACK_DURATION = "duration";

  public static final String BOOK = "Book";
  public static final String BOOK_TITLE = "title";

  public static final String OWNS = "Owns";
  public static final String OWNS_OWNER = "owner";
  public static final String OWNS_ITEM = "item";

  public static final String SELL = "Sell";
  public static final String SELL_BUYER = "buyer";
  public static final String SELL_ITEM = "item";

  // The singleton instance of this ontology
  private static Ontology theInstance = new MusicShopOntology();

  // This is the method to access the singleton music shop ontology object
  public static Ontology getInstance() {
    return theInstance;
  }

  // Private constructor
  private MusicShopOntology() {
    // The music shop ontology extends the basic ontology
    super(ONTOLOGY_NAME, BasicOntology.getInstance())

    try {
      add(new ConceptSchema(ITEM), Item.class);
      add(new ConceptSchema(CD), CD.class);
```

```
      add(new ConceptSchema(TRACK), Track.class);
      add(new ConceptSchema(BOOK), Book.class);
      add(new PredicateSchema(OWNS), Owns.class);
      add(new AgentActionSchema(SELL), Sell.class);

      // Structure of the schema for the Item concept
      ConceptSchema cs = (ConceptSchema) getSchema(ITEM);
      cs.add(ITEM_SERIAL, (PrimitiveSchema) getSchema(BasicOntology.INTEGER),
             ObjectSchema.OPTIONAL); // The serial-number slot is optional and
                                     // allowed values are integers.

      // Structure of the schema for the CD concept
      cs = (ConceptSchema) getSchema(CD);
      cs.addSuperSchema((ConceptSchema) getSchema(ITEM));
      cs.add(CD_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
      cs.add(CD_TRACKS, (ConceptSchema) getSchema(TRACK), 1,
             ObjectSchema.UNLIMITED); // The tracks slot has cardinality > 1

      // Structure of the schema for the Track concept
      cs = (ConceptSchema) getSchema(TRACK);
      cs.add(TRACK_TITLE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
      cs.add(TRACK_DURATION, (PrimitiveSchema)
             getSchema(BasicOntology.INTEGER), ObjectSchema.OPTIONAL);

      // Structure of the schema for the Book concept
      cs = (ConceptSchema) getSchema(BOOK);
      cs.addSuperSchema((ConceptSchema) getSchema(ITEM));
      cs.add(BOOK_TITLE, (PrimitiveSchema) getSchema(BasicOntology.STRING));

      // Structure of the schema for the Owns predicate
      PredicateSchema ps = (PredicateSchema) getSchema(OWNS);
      ps.add(OWNS_OWNER, (ConceptSchema) getSchema(BasicOntology.AID));
      ps.add(OWNS_ITEM, (ConceptSchema) getSchema(ITEM));

      // Structure of the schema for the Sell agent action
      AgentActionSchema as = (AgentActionSchema) getSchema(SELL);
      as.add(SELL_ITEM, (ConceptSchema) getSchema(ITEM));
      as.add(SELL_BUYER, (ConceptSchema) getSchema(BasicOntology.AID));
    }
    catch (OntologyException oe) {
      oe.printStackTrace();
    }
  }
}
```

All the `XXXSchema` classes are included in the `jade.content.schema` package.
From the above code we can see that

- Each schema added to the ontology is associated to a Java class e.g. the schema for the *CD* concept is associated to the `CD.java` class. While using the defined ontology, expressions indicating CDs will be instances of the `CD` class. These Java classes must have a proper structure as described in 4.3.
- Each slot in a schema has a name and a type, i.e. values for that slot must comply with a given schema.

- A slot can be declared as `OPTIONAL` meaning that its value can be `null`. Otherwise a slot is considered `MANDATORY`. If a `null` value for a `MANDATORY` slot is encountered in the validation of a content expression, an `OntologyException` is thrown.
- A slot can have a cardinality > 1, i.e. values for that slot are aggregates of elements of a given type. For example the *tracks* slot in the schema for the *CD* concept can contain 1 or more elements of type *Track*.
- A schema can have a number of super-schemas. This allows defining specialization/extension relationships among concepts. For example the schema of *CD* has the schema of *Item* as a super-type meaning that *CD* is a type of *Item* and therefore that each *CD* instance is also an *Item* instance.

4.3 Developing ontological Java classes

As mentioned in 4.2 each schema included in an ontology is associated with a Java class (or interface). Clearly the structure of these classes must be coherent with the associated schemas. More in details they must obey the following rules.

1) Implementing a proper interface i.e.
   - If the schema is a `ConceptSchema` the class must implement (directly or indirectly) the `Concept` interface.
   - If the schema is a `PredicateSchema` the class must implement (directly or indirectly) the `Predicate` interface.
   - If the schema is a `AgentActionSchema` the class must implement (directly or indirectly) the `AgentAction` interface.

The above interfaces are part of a hierarchy that follows the content reference model presented in 3 and that is included in the `jade.content` package

2) Having the proper inheritance relations i.e.

If S1 is a super-schema of S2 then the class C2 associated to schema S2 must extend the class C1 associated to schema S1.

3) Having the proper member fields and accessor methods i.e.
   - For each slot in schema S1 with name nnn and type (i.e. whose schema is) S2 the class C1 associated to schema S1 must have two accessor methods with the following signature
     ```
     public void setNnn(C2 c);
     public C2 getNnn();
     ```
     where C2 is the class associated to schema S2. In particular if S2 is a schema defined in the BasicOntology then
     - if S2 is the schema for STRING → C2 is java.lang.String
     - if S2 is the schema for INTEGER → C2 is int, long, java.lang.Integer or java.lang.Long[3]
     - if S2 is the schema for BOOLEAN → C2 is boolean or java.lang.Boolean
     - if S2 is the schema for FLOAT → C2 is float, double, java.lang.Float or java.lang.Double
     - if S2 is the schema for DATE → C2 is java.util.Date
     - if S2 is the schema for BYTE_SEQUENCE → C2 is byte[]
     - if S2 is the schema for AID → C2 is jade.core.AID

[3] The user can choose among these options according to his preferences

– For each slot in schema S1 with name nnn, type S2 and cardinality > 1 the class C1 associated to schema S1 must have two accessor methods with the following signature
`public void setNnn(jade.util.leap.List l);`
`public jade.util.leap.List getNnn();`

To exemplify the above rules the classes associated to the CD concept and to the Owns predicate in the music shop example are reported below.

```
// Class associated to the CD schema
package musicShopOntology;

import jade.util.leap.List;

public class CD extends Item { // Note that the Item class (omitted here)
                               // implements Concept
  private String name;
  private List tracks;

  public String getName() {
    return name;
  }
  public void setName(String n) {
    name = n;
  }
  public List getTracks() {
    return tracks;
  }
  public void setTracks(List l) {
    tracks = l;
  }
}

// Class associated to the Owns schema
package musicShopOntology;

import jade.content.Predicate;
import jade.core.AID;

public class Owns implements Predicate
  private AID owner;
  private Item item;

  public AID getOwner() {
    return owner;
  }
  public void setOwner(AID id) {
    owner = id;
  }
  public Item getItem() {
    return item;
  }
  public void setItem(Item i) {
    item = i;
  }
}
```

4.4 Selecting a content language

The `jade.content` package directly includes codecs for two content languages (the SL language and the LEAP language) both supporting the content reference model described in 3. A codec for a content language *L* is a Java object able to manage (see 5.8 for details) content expressions written in the *L* language. In the great majority of the cases a developer can just adopt one of these two content languages and use the related codec without any additional effort. This section gives some hints that can help in choosing which one. If a developer wants his agents to "speak" a different content language he can define a proper codec for it as described in 5.8.

The **SL language** is a human-readable string-encoded (i.e. a content expression in SL is a string) content language and is probably (together with KIF) the mostly diffused content language in the scientific community dealing with intelligent agents. All examples of content expression in this documentation are expressed in SL. In general we suggest to adopt this language especially for agent based applications that are (or can become) open (i.e. where agents from different developer and running on different platforms must communicate). SL includes a number of useful operators such as logical operators (`AND, OR, NOT`) and modal operators (`BELIEF, INTENTION, UNCERTAINTY`). Refer to 5.4.1 for a description of how to use them. Moreover the property of being human-readable can be very helpful when debugging and testing an application.

The **LEAP language** is a non-human-readable byte-encoded (i.e. a content expression in LEAP is a sequence of byte) content language that has been defined ad hoc for JADE within the LEAP project. It is therefore clear that only JADE agents will be able to "speak" the LEAP language. There are some cases however in which the LEAP language is preferable with respect to SL.

- The `LEAPCodec` class is lighter than the `SLCodec` class. When there are strong memory limitations the LEAP language is preferable.
- Unlike the LEAP language, the SL language does not support sequences of bytes.

Finally the developer should take into account that the SL language deals with agent actions particularly. All agent actions in SL must be inserted into the ACTION construct (included in the `BasicOntology` and implemented by the `jade.content.onto.basic.Action` class) that associates the agent action to the `AID` of the agent that is intended to perform the action. Therefore the expression

```
(Sell
      (Book :title "The Lord of the rings")
      (agent-identifier :name Peter)
)
```

cannot be used directly as the content of e.g. a REQUEST message even if it corresponds to an agent action in the Content Reference Model. In fact the SL grammar does not allow it as a first-level expression. The following expression must be used instead

```
(ACTION
      (agent-identifier :name John)
      (Sell
            (Book :title "The Lord of the rings")
            (agent-identifier :name Peter)
      )
)
```

Where John is the agent that is requested to sell the specified book to agent Peter.

4.5 Registering content languages and ontologies to an agent

Before an agent can actually use the defined ontology and the selected content language, they must be registered to the content manager of the agent. This operation is typically (but not necessarily)

performed during agent setup (i.e. in the `setup()` method of the `Agent` class). The following code shows this registration in the case of the Seller agent (the Buyer agent looks like the same) assuming the SL Language is selected.

```
public class SellerAgent extends Agent {
  private Codec codec = new SLCodec();
  private Ontology ontology = MusicShopOntology.getInstance();
  ….
  protected void setup() {
    ….
    getContentManager().registerLanguage(codec);
    getContentManager().registerOntology(ontology)
    ….
  }
  ….
}
```

From now on the content manager will associate the registered `Codec` and `Ontology` objects to the strings returned by their respective `getName()` methods.
Note that, while it is generally a good practice having a singleton `Ontology` object, this is not the case for `Codec` objects as synchronization problems can arise during parsing operations.

4.6 Creating and manipulating content expressions as Java objects

Having defined an ontology (and the classes associated to the types of predicate, agent action and concept it includes), selected a proper language and registered them to the agent's content manager, creating and manipulating content expressions as Java objects is straightforward. The code[4] below shows how the Buyer agent asks the Seller agent if he owns "Synchronicity"[5].

```
….
// Prepare the Query-IF message
ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
msg.addReceiver(sellerAID) // sellerAID is the AID of the Seller agent
msg.setLanguage(codec.getName());
msg.setOntology(ontology.getName());

// Prepare the content. Optional fields are not set
CD cd = new CD();
cd.setName("Synchronicity");
List tracks = new ArrayList();
Track t = new Track();
t.setTitle("Every breath you take");
tracks.add(t);
t = new Track();
t.setTitle("King of pain");
tracks.add(t);
cd.setTracks(tracks);

Owns owns = new Owns();
owns.setOwner(sellerAID);
owns.setItem(cd);
```

[4] This code is likely to be inserted into a proper behaviour implementing a FIPA-Query protocol, but this is out of the scope of this documentation

[5] A very well known CD by The Police

```
try {
  // Let JADE convert from Java objects to string
  getContentManager().fillContent(msg, owns);
  send(msg);
}
catch (CodecException ce) {
  ce.printStackTrace();
}
catch (OntologyException oe) {
  oe.printStackTrace();
}
….
```

In the fillContent() method the Buyer agent's content manager gets the proper `Ontology` and `Codec` objects (on the basis of the values of the `:ontology` and `:language` slots of the `ACLMessage` msg) and let them perform the necessary conversion and check operations. Similarly the Seller agent, receiving the message from the Buyer agent, can handle it as below.

```
….
// Receive the message
MessageTemplate mt = MessageTemplate.and(
    MessageTemplate.MatchLanguage(codec.getName()),
    MessageTemplate.MatchOntology(ontology.getName()) );
ACLMessage msg = blockingReceive(mt);

try {
  ContentElement ce = null;
  if (msg.getPerformative() == ACLMessage.QUERY_IF) {
    // Let JADE convert from String to Java objects
    ce = getContentManager().extractContent(msg);
    if (ce instanceof Owns) {
      Owns owns = (Owns) ce;
      Item it = owns.getItem();
      // Check if I have this item and answer accordingly
      ….
    }
    ….
  }
  catch (CodecException ce) {
    ce.printStackTrace();
  }
  catch (OntologyException oe) {
    oe.printStackTrace();
  }
}
….
```

5 USING THE JADE CONTENT LANGUAGE AND ONTOLOGY SUPPORT. ADVANCED FEATURES

5.1 Combining ontologies

The support for content languages and ontologies included in the `jade.content` package provides an easy way to combine ontologies thus facilitating code re-usage.

In particular it is possible to define that a new ontology extends one or more (previously defined) ontologies by simply specifying the extended ontologies as parameters in the constructor used to create the new ontology.
For instance in the music shop example we included for simplicity all predicates, agent actions and concepts in the `MusicShopOntology`, but as a matter of facts the concept `Item`, the predicate `Owns` and the agent action `Sell` are not strictly related to the music shop domain. They could be included in another more generic ontology called for instance `ECommerceOntology` and the `MusicShopOntology` could be defined as extending the `ECommerceOntology` by adding the `CD`, `Book` and `Track` concepts.
Assuming we moved the above mentioned ontological elements in the `ECommerceOntology`, the `MusicShopOntology` would be modified as below

```
package musicShopOntology;

import jade.content.onto.*;
import jade.content.schema.*;
import eCommerceOntology.*;

public class MusicShopOntology extends Ontology {
  // The name identifying this ontology
  public static final String ONTOLOGY_NAME = "Music-shop-ontology";

  // VOCABULARY
  public static final String CD = "CD";
  public static final String CD_NAME = "name";
  public static final String CD_TRACKS = "tracks";

  public static final String TRACK = "Track";
  public static final String TRACK_TITLE = "title";
  public static final String TRACK_DURATION = "duration";

  public static final String BOOK = "Book";
  public static final String BOOK_TITLE = "title";

  // The singleton instance of this ontology
  private static Ontology theInstance = new MusicShopOntology();

  // This is the method to access the singleton music shop ontology object
  public static Ontology getInstance() {
    return theInstance;
  }

  // Private constructor
  private MusicShopOntology() {
    // The music shop ontology extends the e-commerce ontology
    super(ONTOLOGY_NAME, ECommerceOntology.getInstance())

    try {
      add(new ConceptSchema(CD), CD.class);
      add(new ConceptSchema(TRACK), Track.class);
      add(new ConceptSchema(BOOK), Book.class);

      // Structure of the schema for the CD concept
```

```
cs = (ConceptSchema) getSchema(CD);
cs.addSuperSchema((ConceptSchema) getSchema(EcommerceOntology.ITEM));
cs.add(CD_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
cs.add(CD_TRACKS, (ConceptSchema) getSchema(TRACK), 1,
       ObjectSchema.UNLIMITED); // The tracks slot has cardinality > 1
….
```

It is possible to extend more than one existing ontology by specifying an array of `Ontology` instead of just one `Ontology` object. See the javadoc for more details.

5.1.1 The Vocabulary interface pattern

In the above example of course we must take into account that the `ITEM` constant is defined in `ECommerceOntology`. In cases in which we are dealing with large ontologies that are obtained extending several previously defined ontologies, keeping track of which ontology a given symbol is actually defined in can be quite annoying. To face this problem we suggest the simple design pattern represented in Figure 3.

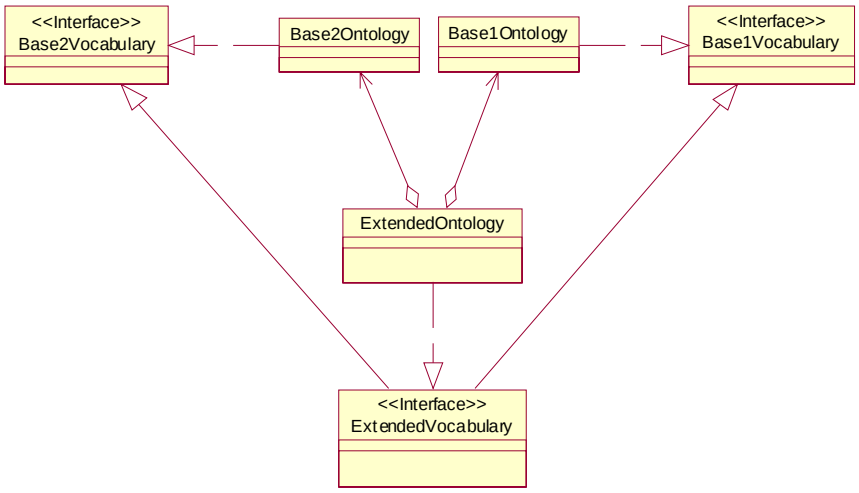

Figure 3The Vocabulary-interface pattern

Where all the constants are defined in the Vocabulary interfaces. By organizing things this way each constant can be accessed as if it were defined in the ExtendedOntology even if it was actually defined in one of the base ontologies.

5.2 Working with abstract descriptors

If using Java objects to represent a content expression (as described in 4) is very convenient for managing the information included in that content expression, there are however some cases where this can create problems.

- Assuming an ontology includes 1000 elements, we need to deal with 1000 classes. Even if these classes can be automatically generated (as described in 6), there can be situations in which dealing with 1000 classes is in any case a problem (for instance if the agent has to be deployed on a small device with memory limitations).
- If multiple inheritance relationships have to be defined between concepts in an ontology (this feature is supported by simply calling several times the `addSuperSchema()` method), the

Java classes representing these concepts must be interfaces as Java does not support multiple inheritance.

- In order to create queries it is necessary to specify variables such as in `QUERY_REF (All ?x (Owns (agent-identifier :name Seller) ?x)` Such a content expression can't be translated into a Java object as an object representing a variable cannot be assigned where an `Item` is required.

For these reasons JADE provides another (less convenient, but more general) way of representing content expressions: each element can be represented as an abstract descriptor that includes

- a ***type-name*** indicating the actual type of the element
- a number of ***named slots*** holding the attributes of the element

This is to say that e.g. the concept `(Person :name Giovanni :age 33)` can be also represented as an instance of the `AbsConcept` class (an abstract descriptor representing a concept) where the `type-name` is set to "`Person`", the `slot` named "`name`" is set to "`Giovanni`"[6] and the `slot` named "`age`" is set to `33`.

At the end of the day there is an abstract descriptor class for each type of element in the content reference model presented in 3 (`AbsPredicate, AbsAgentAction, AbsConcept` …..) and all predicates are represented as instances of `AbsPredicate`, all agent actions are represented as instances of `AbsAgentAction` and so on.

All abstract descriptor classes are included in the `jade.content.abs` package.

Note that primitive values are represented as abstract descriptors too, i.e. as instances of the `AbsPrimitive` class.

As an example, when using abstract descriptors, the code presented in 4.6 to ask the Seller agent if he owns a given CD, would look like (refer to the javadoc for details about the methods provided by the abstract descriptor classes):

```
// Prepare the Query-IF message
ACLMessage msg = new ACLMessage(ACLMessage.QUERY_IF);
msg.addReceiver(sellerAID) // sellerAID is the AID of the Seller agent
msg.setLanguage(codec.getName());
msg.setOntology(ontology.getName());

// Prepare the content. Optional fields are not set
AbsConcept absCd = new AbsConcept(MusicShopOntology.CD);
absCd.set(MusicShopOntology.CD_NAME, "Synchronicity");

AbsAggregate absTracks = new AbsAggregate (BasicOntology.SEQUENCE);
AbsConcept absT = new AbsConcept (MusicShopOntology.TRACK);
absT.set(MusicShopOntology.TRACK_TITLE, "Every breath you take");
absTracks.add(absT);

absT = new AbsConcept (MusicShopOntology.TRACK);
absT.set(MusicShopOntology.TRACK_TITLE, "King of pain");
absTracks.add(absT);

absCd.set(MusicShopOntology.CD_TRACKS, absTracks);

try {
```

[6] More in details it is set to an AbsPrimitive (an abstract descriptor representing a primitive value) representing the string "Giovanni"

```
  // Use the basic ontology to get an abstract descriptor of the Seller AID
  AbsConcept absSeller = ontology.fromObject(sellerAID);

  AbsPredicate absOwns = new AbsPredicate(MusicShopOntology.OWNS);
  absOwns.set(MusicShopOntology.OWNS_OWNER, absSeller);
  absOwns.set(MusicShopOntology.OWNS_ITEM, absCd);
  // Let JADE convert from Abstract descriptor to string
  getContentManager().fillContent(msg, owns);
  send(msg);
}
catch (CodecException ce) {
  ce.printStackTrace();
}
catch (OntologyException oe) {
  oe.printStackTrace();
}
….
```

It should be noted that

- We use the ontology to get an abstract descriptor of the seller AID. This is explained in 5.3.
- In this case we are using an overloaded version of the `fillContent()` method that takes an `AbsContentElement` (instead of a `ContentElement`) as parameter.
- In general the developer can work with both user defined java classes and abstract descriptors depending on the situation. For example java objects can be used normally (as they are more convenient) and abstract descriptors can be used only when dealing with queries (that can't be represented as java objects).
- The `Ontology` class provides an overloaded version of the `add()` method that allows adding a schema to the ontology without specifying any Java class associated to this schema. Clearly if the user adds a schema using this method he will never be able to work with Java objects when dealing with content expressions that refer to that schema.

5.3 The conversion pipeline

Even if the user only works with Java objects, JADE uses the abstract descriptor classes when translating content expressions. More in details when the `fillContent()/extractContent()` methods of the `ContentManager` class are called

- The `Codec` object (associated to the language indicated in the `:language` slot of the message whose content has to be translated) converts a string (or a sequence of byte) into/from an `AbsContentElement`.
- The `AbsContentElement` is validated against its schema.
- The `Ontology` object (associated to the ontology indicated in the `:ontology` slot of the message whose content has to be translated) converts the `AbsContentElement` into/from a Java object of a class implementing the `ContentElement` interface.

Figure 4 graphically represents this conversion pipeline.

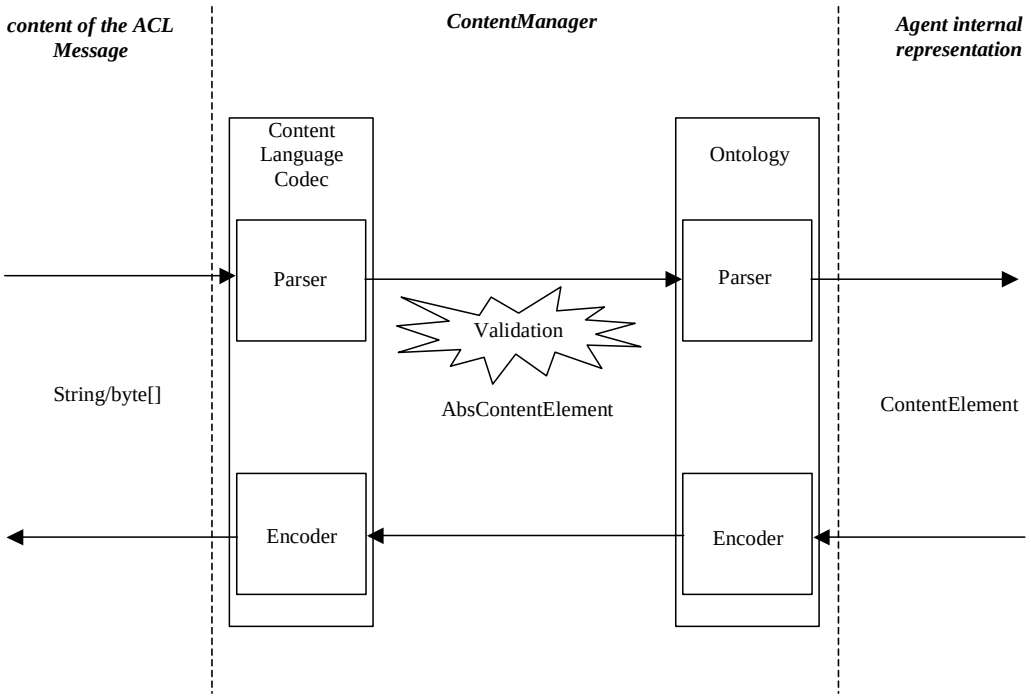

Figure 4The conversion pipeline

The `toObject()` and `fromObject()` methods of the `Ontology` class are used to perform the translation between abstract descriptors and java objects.

5.4 Content language operators

In general a content language, besides defining a proper syntax for content expressions, defines a number of operators such as the logical connectors `AND` and `OR`. Each operator in a content language, like elements in an ontology, can be defined by means of a schema that specifies the structure of all the expressions based on that operator. This schema will be a `PredicateSchema`, an `AgentActionSchema` and so on according to the semantics of the operator. For instance the `AND` operator will be defined by a `PredicateSchema` (since an expression based on the `AND` operator is a logical expression that can be true or false) whose type-name is "AND" and that has two slots:

- `left` whose values must be `Predicate` objects
- `right` whose values must be `Predicate` objects

The operators in a content language therefore form an ontology that can be defined in JADE exactly as user defined ontologies described in 4.2. This ontology can be accessed through the `getInnerOntology()` method of the `Codec` class. It should be noticed however that:

- Unlike user defined ontologies whose elements are in general domain specific, the ontology of the operators of a content language in general only includes domain independent element.
- While user defined ontologies only include schemas of predicates, concepts and agent actions, the ontology of the operators of a content language can also include schemas of other types in the content reference model. For example the `sequence` operator of the SL language if defined by an `AggreateSchema`.

5.4.1 Using the SL operators

The SL language supports a rich set of useful operators ranging from logical operators (`AND, OR, NOT`) to modal operators (`B, I, U, PG`) and action operators (`ACTION, ;, |`). As mentioned in 5.4, these operators are defined by proper schemas. Since there are quite a lot of operators in SL, however, we decided not to provide a Java class for each of them. In particular only the operators that are supported by the SL0 language (`agent-identifier, set, sequence, action, done, result, =`) have a class associated to them. Their schemas and the related Java classes are defined directly in the `BasicOntology` as these operators are necessary to create expressions that are referenced in the definition of the semantics of ACL.
In order to handle expressions that make use of the other operators it is necessary to use the abstract descriptors as described in 5.2. All operator names are available as constants defined in the `jade.content.lang.sl.SLVocabulary` interface.
As an example let's go back to the music shop case and let's assume that the seller agent does not own "Synchronicity". In order to inform the buyer agent about that, the seller can use the `NOT` operator. The code to do that will look like that highlighted in bold.

```
….
// Receive the message
MessageTemplate mt = MessageTemplate.and(
    MessageTemplate.MatchLanguage(codec.getName()),
    MessageTemplate.MatchOntology(ontology.getName()) );
ACLMessage msg = blockingReceive(mt);

try {
  ContentElement ce = null;
  if (msg.getPerformative() == ACLMessage.QUERY_IF) {
    // Let JADE convert from String to Java objects
    ce = getContentManager().extractContent(msg);
    if (ce instanceof Owns) {
      Owns owns = (Owns) ce;
      Item it = owns.getItem();
      if (I don't own the item it) {
        AbsPredicate not = new AbsPredicate(SLVocabulary.NOT);
        not.set(SLVocabulary.NOT_WHAT, ontology.fromObject(owns));
        ACLMessage reply = msg.createReply();
        reply.setPerformative(ACLMessage.INFORM);
        getContentManager().fillContent(reply, not);
        send(reply);
      }
      ….
    }
    ….
  }
  catch (CodecException ce) {
    ce.printStackTrace();
  }
  catch (OntologyException oe) {
    oe.printStackTrace();
  }
}
```

5.5 Creating queries

As already mentioned, working with abstract descriptors is necessary when manipulating queries. In particular two abstract descriptors classes are used when dealing with queries:

- `AbsIRE` - an abstract descriptor representing an Identifying Referential Expression (IRE)
- `AbsVariable` - an abstract descriptor representing a Variable

An IRE always includes a variable and a predicate and therefore the `AbsIRE` class has proper methods to access the `AbsVariable` and the `AbsPredicate` representing the included variable and predicate.

In order to exemplify the usage of `AbsIRE` and `AbsVariable` to create queries let's add now to the `ECommerceOntology` the predicate `Costs` that relates an `Item` with an `integer` representing the price of that item. We also assume to have organized things according to the Vocabulary-interface pattern described in 5.1.1. The definition of this new predicate looks like:

`EcommerceVocabulary` file

```
  …..
  public static final String COSTS = "Costs";
  public static final String COSTS_ITEM = "item";
  public static final String COSTS_PRICE = "price";
  ….
```

`ECommerceOntology` file

```
  ….
  // Private constructor
  private ECommerceOntology() {
    // The e-commerce ontology extends the basic ontology
    super(ONTOLOGY_NAME, BasicOntology.getInstance())

    try {
      add(new ConceptSchema(ITEM), Item.class);
      add(new PredicateSchema(OWNS), Owns.class);
      add(new PredicateSchema(COSTS), Costs.class);
      add(new AgentActionSchema(SELL), Sell.class);
      ……
      // Structure of the schema for the Costs predicate
      PredicateSchema ps = (PredicateSchema) getSchema(COSTS);
      ps.add(COSTS_ITEM, (ConceptSchema) getSchema(ITEM));
      ps.add(COSTS_PRICE, (PrimitiveSchema) getSchema(BasicOntology.INTEGER));
      ……
```

Then we modify the Buyer agent so that it asks the Seller agent for the price of the CD he is interested in before trying to buy it. The code to create the query will look like:

```
// Prepare the Query-REF message
ACLMessage msg = new ACLMessage(ACLMessage.QUERY_REF);
msg.addReceiver(sellerAID) // sellerAID is the AID of the Seller agent
msg.setLanguage(codec.getName());
msg.setOntology(ontology.getName());

// Prepare the content.
try {
  AbsConcept absCd = ontology.fromObject(cd);
  AbsVariable x = new AbsVariable("x", BasicOntology.INTEGER);

  AbsPredicate absCosts = new AbsPredicate(MusicShopOntology.COSTS);
```

```
  absCosts.set(MusicShopOntology.COSTS_ITEM, absCd);
  absCosts.set(MusicShopOntology.COSTS_PRICE, x);

  AbsIRE absIota = new AbsIRE(SLVocabulary.IOTA);
  absIota.setVariable(x);
  absIota.setProposition(absCosts);

  // Let JADE convert from Abstract descriptor to string
  getContentManager().fillContent(msg, absIota);
  send(msg);
}
catch (CodecException ce) {
  ce.printStackTrace();
}
catch (OntologyException oe) {
  oe.printStackTrace();
}
….
```

With reference to the above code it should be noticed that `cd` is the `CD` object we used in 4.6 to ask the Seller agent if he owns “Synchronicity”. Instead of creating an empty `AbsConcept` and filling its slots from scratch (as shown in 5.2) we use the ontology to translate from Java object to abstract descriptor by means of the `fromObject()` method.

Each variable has a name (“x” in this case) and a type indicated as a `String`; in this case the type is `BasicOntology.INTEGER` as the variable is used to replace the price (an integer value) of “Synchronicity”.

Thanks to the Vocabulary-interface pattern we don’t care that the costs predicate is defined in the `ECommerceOntology`, but we just deal with the `MusicShopOntology`.

The “`iota`” operator (indicated by the `SLVocabulary.IOTA` constant) is an operator of the SL language that allows creating IREs indicating “the unique X such that a given predicate (containing the variable X) is `true`”.

5.6 Adding semantic constraints: Facets

The content languages and ontologies support included in the `jade.content` package gives the user the possibility of setting additional constraints (called facets) to the predicates, agent actions and concepts he defines in an ontology. As an example we would like to check that the `price` of an `Item` is always a positive integer and we don’t want to explicitly perform this check each time we deal with the `Costs` predicate. In order to delegate this check to JADE it is sufficient to add to the `price` slot of the schema of the `Costs` predicate a facet that checks that the value of this slot is greater than 0. This facet will be forced each time a content expression including the `Costs` predicate is translated by the `ContentManager`. From the implementation point of view a facet is an instance of a class implementing the `Facet` interface included in the `jade.content.schema` package.

The code to add this facet in the `ECommerceOntology` is

```
……
// Structure of the schema for the Costs predicate
PredicateSchema ps = (PredicateSchema) getSchema(COSTS);
ps.add(COSTS_ITEM, (ConceptSchema) getSchema(ITEM));
ps.add(COSTS_PRICE, (PrimitiveSchema) getSchema(BasicOntology.INTEGER));
ps.addFacet(COSTS_PRICE, new PositiveIntegerFacet());
```

……

Where the `PositiveIntegerFacet` class can be defined as

```
public class PositiveIntegerFacet implements Facet {
  void validate(AbsObject abs, Ontology onto) throws OntologyException {
    try {
      AbsPrimitive p = (AbsPrimitive) abs;
      if (p.getInteger() <= 0) {
        throws new OntologyException("Integer value <= 0");
      }
    }
    catch (Exception e) {
      throws new OntologyException("Not an Integer value", e);
    }
  }
}
```

5.7 Disabling semantic checks to improve performances

As described in 5.3, when translating a content expression form a string/byte sequence representation to a Java object representation (and vice-versa), JADE validates the content expression against its schema. If this validation succeeds the content is semantically correct with respect to the ontology it refers to and the agent does not need to perform any explicit check. It's clear however that the validation process takes its time. In order to speed up the performances the user can disable the validation process by means of the `setValidationMode()` of the `ContentManager` class.
This can be a useful trick when developing "closed" applications (i.e. applications where all agents are JADE agents and all of them use the JADE content languages and ontologies support). Unless there are design/implementation bugs in facts, in these cases content expressions exchanged in agent communication are likely consistent. On the other hand it is strongly suggested to keep the validation process enabled when dealing with "open" applications in which different agents can have completely different ways of internally representing content expressions.

5.8 User-defined content languages

As mentioned in 4.4, the `jade.content` package directly includes codecs for two content languages (the SL language and the LEAP language) both supporting the content reference model described in 3. In the great majority of the cases a developer can just adopt one of these two content languages and use the related codec without any additional effort. There are cases however in which the user is forced to create agents "speaking" a different content language and therefore he has to develop an ad-hoc codec for that language.
From the implementation point of view a content language codec in JADE is an instance of a class extending the `jade.content.lang.Codec` abstract class and, in particular, implementing the two methods `decode()` and `encode()` to respectively

- parse the content in an ACL message and convert it into an `AbsContentElement` object.
- encode the content from an `AbsContentElement` object into the content language syntax and encoding.

More in details, as a content expression inside an `ACLMessage` can be represented as a `String` or as a sequence of byte (i.e. as a `byte[]`), the `Codec` class is further specialized into `StringCodec` and `ByteArrayCodec`. These two classes differ in the signatures of the abstract methods `decode()`

and `encode()`: In the former they translate a `String` into/from an `AbsContentElement`, while in the latter they translate a `byte[]` into/from an `AbsContentElement`.
As an example the `SLCodec` (included in the `jade.content.lang.sl package`) extends `StringCodec`, while the `LEAPCodec` (included in the `jade.content.lang.leap` package) extends `ByteArrayCodec`.

5.9 Introspectors

In order to decouple the definition of an ontology (i.e. the definitions of the schemas in the ontology) form the implementation of the classes representing the predicates, agent actions and concepts included in the ontology, the `Ontology` class does not deal directly with the translation from abstract descriptors to java objects. On the other hand it delegates that task to an internal object that implements the `Introspector` interface included in the `jade.content.onto` package.
The `Introspector` interface includes three methods:
`externalize()` to translate a Java object into an abstract descriptor
`internalize()` to translate an abstract descriptor into a Java object
`checkClass()` to check that a Java class associated to a schema has a structure (i.e. accessor methods) consistent with that schema and therefore that successive calls to the `externalize()` and `internalize()` methods will succeed.
When defining an ontology it is possible to specify the `Introspector` object that the ontology will use by passing it as a parameter in the constructor of the ontology. The `jade.content` package comes with a default `Introspector` (called `ReflectiveIntrospector` as it uses Java Reflection to perform the translations) that is used unless the user explicitly specify a different one and that requires the ontological Java classes to have the structure described in 4.3. By using a different `Introspector` it is therefore possible to use Java classes that does not meet the structure described in 4.3 to represent the predicates, agent actions and concepts included in an ontology.

6 USING PROTÉGÉ TO CREATE JADE ONTOLOGIES

Especially when dealing with large ontologies, developing the ontology definition class (i.e. the schemas) and the Java classes representing the predicates, agent actions and concepts included in the ontology “by hand” as described in 4.2 and 4.3 can be really time consuming. Thanks to a proper plug-in (called **beangenerator**) implemented by C.J. van Aart from Department of Social Science Informatics (SWI) University of Amsterdam, it is possible to define the ontology using Protégé and then let the beangenerator automatically create the ontology definition class and the predicates, agent actions and concepts classes.
This process is very convenient as it allows working with an ad-hoc graphical tool (as Protégé is) instead of writing Java code in the ontology definition process.
The beangenerator can be freely downloaded from
http://www.swi.psy.uva.nl/usr/aart/beangenerator together with detailed instructions about how to plug it into Protégé and how to use it to convert a Protégé ontology into a JADE ontology.

7 MIGRATING FROM THE OLD SUPPORT TO THE NEW ONE

This section briefly describes how to modify code already written for the “old” support for content languages and ontologies provided by JADE so that it can be used with the new one.
There are four differences that must be considered.

1) The format of the ontology definition class is different. This is certainly the main effort that is required in this process as it basically requires re-writing the ontology definition class from scratch as described in 4.2. It should be noticed however that this modification is completely self-contained i.e. it does not require any modification to the application code that uses the ontology.

2) The structure of the Java classes representing the predicates, agent actions and concepts included in the ontology are different. In the old support in facts, given a slot named nnn of type T with cardinality > 1 the associated Java class should have two accessor methods with the signature

```
public void addNnn(T t);
public Iterator getAllNnn();
```

On the other hand in the new support the two accessor methods should be

```
public void setNnn(List l);
public List getNnn();
```

In order not to force the user to change all these accessor methods (note that this modification would also require changing the application code that uses the ontology) a proper `Introspector` (see 5.9 for details) is provided that allows using classes with the "old" structure within the "new" support. This `Introspector` is called `BCReflectiveIntrospector` (Backward-Compatible Reflective Introspector) and is included in the `jade.content.onto` package.
At the end of the day the user only need to specify that the ontology must use a `BCReflectiveIntrospector` instead of the default `ReflectiveIntrospector`.

3) All Java classes representing predicates, agent actions and concepts in the ontology must be modified to implement `Predicate`, `AgentAction` and `Concept` respectively.

4) When registering the `Codec` and the `Ontology` and when filling/extracting ACLMessages content the methods of the `ContentManager` must be used instead of the respective methods of the `Agent` class.