GATE — An Application Developer's Guide

Valentin Tablan
Diana Maynard
Kalina Bontcheva
Hamish Cunningham

Department of Computer Science
University of Sheffield, UK

V.Tablan, D.Maynard, K.Bontcheva,
H.Cunningham@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/~valyt,diana,kalina,hamish

19 July 2004

Contents

1 Introduction **3**

2 Gazetteers **4**

2.1 Introduction 4

2.2 Creating and Modifying Gazetteer Lists 5

2.3 Using Gaze 5

3 Introduction to JAPE **7**

3.1 Introduction 7

3.2 Pattern matching 7

3.2.1 Context 8

3.2.2 Macros 9

3.3 Simple Pattern actions 9

3.4 Setting the options 10

3.5 Use of priority 11

3.6 Using phases sequentially 11

3.7 Some tricks 12

3.7.1 Negative operator 12

3.7.2 Matching special characters 13

4 API **14**

4.1 Resource Management in GATE 14

4.2 Language Resources 16

4.2.1 GATE Documents 16

4.2.2 Feature Maps 17

4.2.3 Annotation Sets 18
4.2.4 Annotations 19
4.2.5 GATE Corpora 22
4.3 Ontologies 23
4.4 Processing Resources 23
4.5 Controllers 25
4.6 Persistent Applications 26

5 Using Java for JAPE **28**
5.1 Introduction 28
5.1.1 A more complex example 30
5.2 Adding a feature to the document 31
5.3 Finding the Tokens of a matched annotation 32

6 Tools for Annotation and Evaluation **35**
6.1 Manual annotation 35
6.1.1 Simple method of adding annotations 35
6.1.2 Complex method of adding annotations 35
6.1.3 Modifying and removing annotations 36
6.2 Evaluation 36
6.2.1 Annotation Diff 36
6.2.2 Corpus Benchmark tool 37
6.2.3 How to define the properties of the benchmark tool . . 38

Chapter 1

Introduction

This report provides a concise guide to developing NLP applications with GATE, with focus on using the Gate API. Programming examples are provided to clarify the functionality of the API, e.g., as part of JAPE grammars or Java processing resources.

For an introduction to the GATE graphical user interface and general background on using GATE, please refer to the Gate User Guide.

Chapter 2

Gazetteers

2.1 Introduction

The gazetteer consists of a set of lists containing names of things such as cities, organisations, days of the week, etc. These lists are typically used to assist with the task of named entity recognition, although they may be used for any purpose. When the gazetteer is run on a document, annotations of type Lookup will be created for each matching string in the text. The gazetteer does not depend on Tokens or on any other annotation. This means that an entry may span more than one token. A Lookup annotation will only be created if the entire entry is matched. Partial entries are not matched. An entry does not have to correspond with a Token annotation to match, but it must be delimited by white space or punctuation.

Each list is a plain text file, with one entry per line.

Below is a section of the list for units of currency:

```
Ecu
European Currency Units
FFr
Fr
German mark
German marks
New Taiwan dollar
New Taiwan dollars
NT dollar
NT dollars
```

An index file (usually called lists.def) is used to access these lists. Each gazetteer list should reside in the same directory as the index file. For each

list, a major type must be specified and, optionally, a minor type. In the example below, the first column refers to the list name, the second column to the major type, and the third to the minor type. These lists are compiled into finite state machines. Any text strings matched by these machines will be annotated with features specifying the major and minor types.

```
currency_prefix.lst:currency_unit:pre_amount
currency_unit.lst:currency_unit:post_amount
date.lst:date:specific_date
day.lst:date:day
month.lst:date:month
season.lst:date:season
```

Grammar rules can specify the types to be identified in particular circumstances. The major and minor types enable this identification to take place, by giving access to items stored in particular lists or combinations of lists.

For example, if a day needs to be identified, the minor type "day" would be specified in the grammar, in order to match only information about specific days. If any kind of date needs to be identified, the major type "date" would be specified. This might include weeks, months, years etc. as well as days of the week, and would give access to all the items stored in day.lst, month.lst, season.lst, and date.lst in the example shown.

2.2 Creating and Modifying Gazetteer Lists

Gazetteer lists can be modified using any text editor. Use of the GATE Unicode editor is advised, however, in order to ensure that the lists are stored as UTF-8, which will minimise any language encoding problems, particulary if e.g. accents are present.

To create a new list, simply add an entry for that list to the definitions file and add the new list in the same directory as the existing lists.

After any modifications have been made, ensure that you reinitalise the gazetteer PR in GATE, if one is already loaded, before rerunning your application.

2.3 Using Gaze

There is also a gazetteer viewer and editor available within GATE, if the correct creole resource is present. If it is available, double clicking on the

gazetteer list from the resource tree will display the contents of the gazetteer in the main window. The first pane will display the definition file, while the right pane will display whichever gazetteer list has been selected from it.

A gazetteer list can be modified simply by typing in it. it can be saved by clicking the Save button. When a list is saved, the whole gazetteer is automatically reinitialised (and will be ready for use in GATE immediately).

To edit the definition file, right click inside the pane and choose from the options (Inset, Edit, Remove). A pop-up menu will appear to guide you through the remaining process. Save the definition file byu selecting Save. Again, the gazetteer will be reinitialised automatically.

Chapter 3

Introduction to JAPE

3.1 Introduction

JAPE allows you to recognise regular expressions in annotations on documents. A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The phases run sequentially and constitute a cascade of finite state transducers over annotations. The left-hand-side (LHS) of the rules consist of an annotation pattern that may contain regular expression operators (e.g. *, ?, +). The right-hand-side (RHS) consists of annotation manipulation statements. Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements.

3.2 Pattern matching

A pattern is specified in terms of one or more annotations, and optionally, the values of any or all of its features. The following operators can be used:

- | - or
- * - zero or more occurrences
- ? - zero or one occurrences
- + - one or more occurrences

Note that there is no negative operator. There are, however, ways round this problem, as will be explained later.

An operator can operate on any pattern enclosed in round brackets. Every complete pattern to be annotated must be enclosed in round brackets and followed by a label. A label is denoted by a preceding semi-colon. In the example below, the label is :loc.

```
(
 {Lookup.majorType == location}
 )
 :loc
```

It is possible to have more than one pattern (and corresponding label) on the LHS of a rule, e.g.

```
(
  {Lookup.majorType == jobtitle}
 ):jobtitle
 (
  {TempPerson}
 ):person
```

Nested patterns are also permitted (if correctly labelled), e.g.

```
(
 (
  {Lookup.majorType == jobtitle}
 ):jobtitle

 {TempPerson}
):person
```

3.2.1 Context

Context before or after a pattern can also be indicated by enclosing it in a set of round brackets. The difference between a pattern to be annotated and context not to be annotated is simply the presence of a label, i.e. an unlabelled pattern acts as context. For example, the following rule would annotate the pattern matched by YEAR if the words “in” or “by” preceded it. Note that context cannot be reused in more than one rule in the same grammar phase, i.e. it is always consumed if the rule fires.

```
Rule: YearContext1
```

```
 ({Token.string == "in"}|
  {Token.string == "by"}
 )
 (YEAR)
 :date
```

3.2.2 Macros

Macros can also be used in the LHS of rules. This means that instead of expressing the information in the rule, it is specified in a macro, which can then be called in the rule. The reason for this is simply to avoid having to repeat the same information in several rules. Macros can themselves be used inside other macros. A macro used in one phase of a grammar does not need to be respecified in a later phase, i.e. a rule may call a macro specified in a previous phase of the same grammar.

The example below shows a TITLE macro which is then used in a rule. Conventionally, macros are written in capital letters, though this is not essential. When used in the rule, a macro must not be enclosed in curly braces, but only (optionally) in round brackets.

```
Macro: TITLE
(
 {Title}
 ({Token.string == "."})?
 ({Title})?
 ({Token.string == "."})?
)

Rule: TitlePerson
(
 (TITLE)
 ({Upper})+
):person
-->
...
```

3.3 Simple Pattern actions

The RHS of the rule contains information about the annotation to be given to the pattern. Information about the pattern is transferred from the LHS of the rule using the label. and annotated with the entity type (which

follows it). Finally, features and their corresponding values are added to the annotation.

In the example rule below, the label is "loc". The RHS of the rule is the part which follows the arrow. The label is transferred to the RHS of the rule and the annotation type "Location" is added to the pattern. The annotation is given two features, "kind" and "rule" with the values "city" and "GazCity" respectively. Both of these features are optional. The first is used to give us more specific information about the annotation, i.e. that it is a particular kind of Location. The second is used mainly for debugging purposes, so that we can keep track of which rule fired. The resulting annotation and its features and values will all be displayed, together with the string and the offsets, in the annotations tablein the GATE GUI (assuming that the grammar has been run on a document and the rule has been fired).

```
Rule: GazCity
 (
 {Lookup.majorType == city}
 )
 :loc -->
  :loc.Location = {kind="city", rule=‘‘GazCity’’}
```

3.4 Setting the options

At the beginning of each grammar, several options can be set:

Control - this defines the method of rule matching. More on this later.

Debug - when set to true, if the grammar is running in Appelt mode and there is more than one possible match, the conflicts will be displayed in the messages window.

Input annotations must also be defined at the start of each grammar. If no annotations are defined, the default will be Token, SpaceToken and Lookup (i.e. only these annotations will be considered when attempting a match). Every annotation type that is going to be matched in that grammar phase must be included in the Input set. Any annotation type that is not defined will be ignored in the pattern matching. This is very useful in the case of white space. If SpaceTokens are specified, then every possible space between tokens in patterns must be described in the rules. If they are not specified, then this is not necessary, as their presence or absence will be ignored.

3.5 Use of priority

Each grammar phase has 3 possible control styles: “brill”, “first” and “appelt”. This is specified at the beginning of the grammar. For named entity recognition, the appelt style is generally the most suitable. When writing JAPE grammars for other uses, the other styles are often useful.

The “brill” style means that when more than one rule matches the same region of the document, they are all fired. The result of this is that a segment of text could be allocated more than one entity type, and that no proiority ordering is necessary.

With the “first” style, a rule fires for the first match that is found. This makes it unappropiate for rules that end in ”+” or ”?” or ”*”. Once a match is found, the rule is fired; it does not attempt to get a longer match (as the other two styles do).

With the “appelt” style, only one rule can be fired for the same pattern, according to a set of priority rules. Priority operates in the following way.

1. From all the rules that match a region of the document starting at some point X, the one which matches the longest region is fired.
2. If more than one rule matches the same region, the one with the highest priority is fired.
3. If there is more than one rule with the same priority, the one defined last in the grammar is fired.

An optional priority declaration is associated with each rule, which should be a positive integer. The higher the number, the greater the priority. By default (if the priority declaration is missing) all rules have the priority -1 (i.e. the lowest priority).

3.6 Using phases sequentially

A JAPE grammar consists of a set of sequential phases. The list of phases is specified (in the order in which they are to be run) in a file, conventionally named main.jape. When loading the grammar into GATE, it is only necessary to load this main file – the phases will then be loaded automatically. It is, however, possible to omit this main file, and just load the phases individually, but this is much more time-consuming. The grammar phases do not need to be located in the same directory as the main file, but if they are not, the relative path should be specified for each phase.

One of the main reasons for using a sequence of phases is that a pattern can only be used once in each phase, but it can be reused in a later phase. Combined with the fact that priority can only operate within a single grammar, this can be exploited to help deal with ambiguity issues.

The solution currently adopted is to write a grammar phase for each annotation type, or for each combination of similar annotation types, and to create temporary annotations. These temporary annotations are accessed by later grammar phases, and can be manipulated as necessary to resolve ambiguity or to merge consecutive annotations. The temporary annotations can either be removed later, or left and simply ignored.

Generally, annotations about which we are more certain are created earlier on. Annotations which are more dubious may be created temporarily, and then manipulated by later phases as more information becomes available.

3.7 Some tricks

Although the JAPE language has some limitations as to how rules and patterns can be expressed, there are some useful tricks to overcome these problems.

3.7.1 Negative operator

A negative operator cannot be specified as such. One solution to this is to create a "negative rule" which has higher priority than the matching "positive rule". The style of matching must be Appelt for this to work. To create a negative rule, simply state on the LHS of the rule the pattern that should NOT be matched, and on the RHS do nothing. In this way, the positive rule cannot be fired if the negative pattern matches, and vice versa, which has the same end result as using a negative operator. A useful variation for developers is to create a dummy annotation on the RHS of the negative rule, rather than to do nothing, and to give the dummy annotation a rule feature. In this way, it is obvious that the negative rule has fired. Alternatively, use Java code on the RHS to print a message when the rule fires. An example of a matching negative and positive rule follows. Here, we want a rule which matches a surname followed by a comma and a set of initials. But we want to specify that the initials shouldn't have the POS category PRP (personal pronoun). So we specify a negative rule that will fire if the PRP category exists, thereby preventing the positive rule from firing.

```
Rule: NotPersonReverse
Priority: 20
// we don't want to match ''Jones, I''

( {Token.category == NNP}
  {Token.string == ","}
  {Token.category == PRP}
):label
-->
{}

Rule:   PersonReverse
Priority: 5
// we want to match ``Jones, F.W.''

(
 {Token.category == NNP}
 {Token.string == ","}
 (INITIALS)?
)
:person -->
 :person.Person = {rule = ``PersonReverse''}
```

3.7.2 Matching special characters

To specify a single or double quote as a string, precede it with a backslash, e.g.

```
{Token.string=="\""}
```

will match a double quote. For other special characters, such as "$", enclose it in double quotes, e.g.

```
{Token.category == "PRP$"}
```

.

Chapter 4

API

GATE defines three different types of resources:

Language Resources : (**LRs**) entities that hold linguistic data.

Processing Resources : (**PRs**) entities that process data.

Visual Resources : (**VRs**) components used for building graphical interfaces.

These resources are collectively named **CREOLE**[1] resources.

All CREOLE resources have some associated meta-data in the form of an entry in a special XML file named `creole.xml`. The most important role of that meta-data is to specify the set of parameters that a resource understands, which of them are required and which not, if they have default values and what those are.

All resource types have creation-time parameters that are used during the initialisation phase. Processing Resources also have run-time parameters that get used during execution (see section 4.4 for more details).

Controllers are used to define GATE applications and have the role of controlling the execution flow (see section 4.5 for more details).

4.1 Resource Management in GATE

This section describes how to create and delete CREOLE resources as objects in a running Java virtual machine. This process involves using GATEs Factory class[2], and, in the case of LRs, may also involve using a DataStore.

[1] CREOLE stands for Collection of REusable Objects for Language Engineering

[2] Fully qualified name: `gate.Factory`

CREOLE resources are Java Beans; creation of a resource object involves using a default constructor, then setting parameters on the bean, then calling an `init()` method. The Factory takes care of all this, makes sure that the GUI is told about what is happening (when GUI components exist at runtime), and also takes care of restoring LRs from DataStores. **A programmer using GATE should never call the constructor of a resource: always use the Factory!**

Creating a resource involves providing the following information:

- **fully qualified class name** for the resource. This is the only **required** value. For all the rest, defaults will be used if actual values are not provided.
- values for the **creation time parameters**.†
- initial values for **resource features**.† For an explanation on features see section 4.2.2.
- a **name** for the new resource;

† Parameters and features need to be provided in the form of a GATE Feature Map which is essentially a java Map (`java.util.Map`) implementation, see section 4.2.2 for more details on Feature Maps.

e.g. Here is an example that creates a GATE document:

```
URL u = new URL("http://gate.ac.uk/");
FeatureMap params = Factory.newFeatureMap();
params.put("sourceUrl", u);
FeatureMap features = Factory.newFeatureMap();

Document doc = (Document)
  Factory.createResource("gate.corpora.DocumentImpl",
                         params, features, "GATE Homepage");
```

Apart from `createResource()` methods with different signatures, `Factory` also provides some shortcuts for common operations:

`newFeatureMap()` creates a new Feature Map (as used in the example above).

`newDocument(String content)` creates a new GATE Document starting from a String value that will be used to generate the document content.

`newDocument(URL sourceUrl)` creates a new GATE Document using the text pointed by an URL to generate the document content.

`newDocument(URL sourceUrl, String encoding)` same as above but allows the specification of an encoding to be used while downloading the document content.

`newCorpus(String name)` creates a new GATE Corpus with a specified name.

GATE maintains various data structures that allow the retrieval of loaded resources. When a resource is no longer required, it needs to be removed from those structures in order to remove all references to it, thus making it a candidate for garbage collection. This is achieved using the `deleteResource(Resource res)` method on `Factory`.

Simply removing all references to a resource from the user code will **NOT** be enough to make the resource collect-able. Not calling `Factory.deleteResource()` **will** lead to memory leaks!

4.2 Language Resources

This section describes the implementation of documents and corpora in GATE.

4.2.1 GATE Documents

Documents are modelled as content plus annotations (see section 4.2.4) plus features (see section 4.2.2).

The content of a document can be any implementation of the `gate.DocumentContent` interface; the features are <attribute, value> pairs stored a Feature Map. Attributes are String values while the values can be any Java object.

The annotations are grouped in sets (see section 4.2.3). A document has a default (anonymous) annotations set and any number of named annotations sets.

Documents are defined by the `gate.Document` interface and the provided implementations are:

`gate.corpora.DocumentImpl` : transient document. Can be stored persistently through Java serialisation.

`gate.corpora.DatabaseDocumentImpl` : database persistent documents. It can be used in conjunction with database datastores.[3]

Main Document functions are presented in table 4.1.

Content Manipulation	
Method	**Purpose**
`DocumentContent` **getContent**`()`	Gets the Document content.
`void` **edit**`(Long start, Long end, DocumentContent replacement)`	Modifies the Document content.
`void` **setContent**`(DocumentContent newContent)`	Replaces the entire content.
Annotations Manipulation	
Method	**Purpose**
`public AnnotationSet` **getAnnotations**`()`	Returns the default annotation set.
`public AnnotationSet` **getAnnotations**`(String name)`	Returns a *named* annotation set.
`public Map` **getNamedAnnotationSets**`()`	Returns *all* the named annotation sets.
`void` **removeAnnotationSet**`(String name)`	Removes a named annotation set.
Input Output	
`String` **toXml**`()`	Serialises the Document in XML format.
`String` **toXml**`(Set aSourceAnnotationSet, boolean includeFeatures)`	Generates XML from a set of annotations only, trying to preserve the original format of the file used to create the document.

Table 4.1: `gate.Document` methods.

4.2.2 Feature Maps

All CREOLE resources as well as the *Controllers* and the annotations can have attached meta-data in the form of *Feature Maps*.

A Feature Map is a Java Map (i.e. it implements the `java.util.Map` interface) and holds <attribute-name, attribute-value> pairs. The attribute names are Strings while the values can be any Java Objects.

[3]Currently implementations for ORACLE and PostgreSQL are provided.

The use of non-*Serialisable* objects as values is strongly discouraged.

Feature Maps are created using the `gate.Factory.newFeatureMap()` method.

The actual implementation for FeatureMaps is provided by the `gate.util.SimpleFeatureMapImpl` class.

Objects that have features in GATE implement the `gate.util.FeatureBearer` interface which has only the two accessor methods for the object features: `FeatureMap getFeatures()` and `void setFeatures(FeatureMap features)`.

e.g. *How to get a particular feature from an object?*

```
Object obj;
String featureName = "length";
if(obj instanceof FeatureBearer){
  FeatureMap features = ((FeatureBearer)obj).getFeatures();
  Object value = (features == null) ? null :
                                      features.get(featureName);
}
```

4.2.3 Annotation Sets

A GATE document can have one or more annotation layers — an anonymous one, (also called *default*), and as many *named* ones as necessary.

An annotation layer is organised as a *Directed Acyclic Graph (DAG)* on which the nodes are particular locations —*anchors*— in the document content and the arcs are made out of annotations reaching from the location indicated by the start node to the one pointed by the end node (see Figure 4.1 for an illustration). Because of the *graph* metaphor, the annotation layers are also called *annotation graphs*. In terms of Java objects, the annotation layers are represented using the *Set* paradigm as defined by the collections library and they are hence named *annotation sets*. The terms of annotation *layer*, *graph* and *set* are interchangeable and refer to the same concept when used in this guide.

An annotation set holds a number of annotations and maintains a series of indices in order to provide fast access to the contained annotations.

The GATE Annotation Sets are defined by the `gate.AnnotationSet` interface and the current implantations are:

`gate.annotation.AnnotationSetImpl` annotation set implementation used by transient documents.

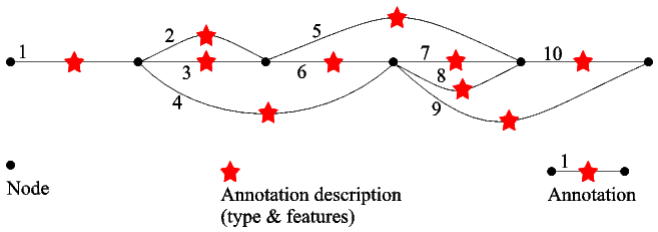

Figure 4.1: The Annotation Graph model.

`gate.annotation.DatabaseAnnotationSetImpl` annotation set implementation used by persistent documents.

The annotation sets are created by the document as required. The first time a particular annotation set is requested from a document it will be transparently created if it doesn't exist.

Tables 4.2 and 4.3 list the most used Annotation Set functions.

e.g. *How to iterate from left to right over all annotations of a given type?*

```
AnnotationSet annSet = ...;
String type = "Person";
//Get all person annotations
AnnotationSet persSet = annSet.get(type);
//Sort the annotations
List persList = new ArrayList(persSet);
Collections.sort(persList, new gate.util.OffsetComparator());
//Iterate
Iterator persIter = persList.iterator();
while(persIter.hasNext()){
...
}
```

4.2.4 Annotations

An **annotation**, is a form of meta-data attached to a particular section of document content. The connection between the annotation and the content it refers to is made by means of two pointers that represent the start and end locations of the covered content. An annotation must also have a type (or a name) which is used to create classes of similar annotations, usually linked together by their semantics.

Annotations Manipulation	
Method	**Purpose**
`Integer` **add**`(Long start, Long end, String type, FeatureMap features)`	Creates a new annotation between two offsets, adds it to this set and returns its id.
`Integer` **add**`(Node start, Node end, String type, FeatureMap features)`	Creates a new annotation between two nodes, adds it to this set and returns its id.
`boolean` **remove**`(Object o)`	Removes an annotation from this set.
Nodes	
Method	**Purpose**
`Node` **firstNode**`()`	Gets the node with the smallest offset.
`Node` **lastNode**`()`	Gets the node with the largest offset.
`Node` **nextNode**`(Node node)`	Get the first node that is relevant for this annotation set and which has the offset larger than the one of the node provided.
Set implementation	
`Iterator` **iterator**`()`	
`int` **size**`()`	

Table 4.2: `gate.AnnotationSet` methods (general purpose).

Searching	
`AnnotationSet` **get**`(Long offset)`	Select annotations by offset. This returns the set of annotations whose start node is the least such that it is less than or equal to offset. If a positional index doesn't exist it is created. If there are no nodes at or beyond the offset parameter then it will return null.
`AnnotationSet` **get**`(Long startOffset, Long endOffset)`	Select annotations by offset. This returns the set of annotations that overlap totally or partially with the interval defined by the two provided offsets. The result will include all the annotations that either: • start before the start offset and end strictly after it • start at a position between the start and the end offsets
`AnnotationSet` **get**`(String type)`	Returns all annotations of the specified type.
`AnnotationSet` **get**`(Set types)`	Returns all annotations of the specified types.
`AnnotationSet` **get**`(String type, FeatureMap constraints)`	Selects annotations by type and features.
`Set` **getAllTypes**`()`	Gets a set of java.lang.String objects representing all the annotation types present in this annotation set.

Table 4.3: `gate.AnnotationSet` methods (searching).

An Annotation is defined by:

start node a location in the document content defined by an offset.

end node a location in the document content defined by an offset.

type a String value.

features (see section 4.2.2).

ID an Integer value. All annotations IDs are unique inside an annotation set.

In GATE, annotations are defined by the `gate.Annotation` interface and implemented by the `gate.annotation.AnnotationImpl` class. Annotations exist only as members of annotation sets (see section 4.2.3) and they should not be directly created by means of a constructor. Their creation should always be delegated to the containing annotation set.

4.2.5 GATE Corpora

A corpus in GATE is a Java List (i.e. an implementation of `java.util.List`) of documents. GATE corpora are defined by the `gate.Corpus` interface and the following implementations are available:

`gate.corpora.CorpusImpl` used for transient corpora.

`gate.corpora.SerialCorpusImpl` used for persistent corpora that are stored in a serial datastore (i.e. as a directory in a file system).

`gate.corpora.DatabaseCorpusImpl` used for persistent corpora stored in a database datastore.

Apart from implementation for the standard List methods, a Corpus also implements the methods in table 4.4.

e.g. *How to create a corpus from all XML files in a directory?*

```
Corpus corpus = Factory.newCorpus("My XML Files");
File directory = ...;
java.io.FileFilter filter = new gate.util.ExtensionFileFilter();
filter.addExtension("xml");
URL url = directory.toURL();
corpus.populate(url, filter, null, false);
```

Method	Purpose
`String` **getDocumentName**`(int index)`	Gets the name of a document in this corpus.
`List` **getDocumentNames**`()`	Gets the names of all the documents in this corpus.
`void` **populate**`(URL directory, FileFilter filter, String encoding, boolean recurseDirectories)`	Fills this corpus with documents created on the fly from selected files in a directory. Uses a `FileFilter` to select which files will be used and which will be ignored. A simple file filter based on extensions is provided in the Gate distribution (`gate.util.ExtensionFileFilter`).

Table 4.4: `gate.Corpus` methods.

4.3 Ontologies

Starting from GATE version `3.1`, support for ontologies has been added. Ontologies are nominally Language Resources but are quite different from documents and corpora and are detailed in this section dedicated to them.

Classes related to ontologies are to be found in the `gate.creole.ontology` package and its sub-packages. The top level package defines an abstract API for working with ontologies while the sub-packages contain concrete implementations. The GATE API for ontologies does not provide full support for all operations that are posible using different ontology models (like RDF-S, OWL or DAML); it only tries to provide support for all the operations that all these formalisms have in common. This includes hierarchies of classes, instances (or individuals) and properties. The terminology used in the naming of the classes and the operations is sligthly biased toward RDF-S and OWL.

The entry point to the ontology API is the `gate.creole.ontology.Ontology` interface which is the base interface for all concrete implementations. It provides methods for accessing the class hierarchy, listing the instances and the properties.

4.4 Processing Resources

Processing Resources (**PRs**) represent entities that are primarily algorithmic, such as parsers, generators or ngram modellers.

They are created using the GATE Factory in manner similar the Language Resources. Besides the creation-time parameters they also have a set of run-time parameters that are set by the system just before executing them.

Analysers are a particular type of processing resources in the sense that they always have a `document` and a `corpus` among their run-time parameters.

The most used methods for Processing Resources are presented in table 4.5

Method	**Purpose**
`void` **setParameterValue**`(String paramaterName, Object parameterValue)`	Sets the value for a specified parameter. method inherited from `gate.Resource`
`void` **setParameterValues**`(FeatureMap parameters)`	Sets the values for more parameters in one step. method inherited from `gate.Resource`
`Object` **getParameterValue**`(String paramaterName)`	Gets the value of a named parameter of this resource. method inherited from `gate.Resource`
`Resource` **init**`()`	Initialise this resource, and return it. method inherited from `gate.Resource`
`void` **reInit**`()`	Reinitialises the processing resource. After calling this method the resource should be in the state it is after calling init. If the resource depends on external resources (such as rules files) then the resource will re-read those resources. If the data used to create the resource has changed since the resource has been created then the resource will change too after calling reInit().
`void` **execute**`()`	Starts the execution of this Processing Resource.
`void` **interrupt**`()`	Notifies this PR that it should stop its execution as soon as possible.
`boolean` **isInterrupted**`()`	Checks whether this PR has been interrupted since the last time its Executable.execute() method was called.

Table 4.5: `gate.ProcessingResource` methods.

4.5 Controllers

Controllers are used to create GATE Applications. A Controller handles a set of Processing Resources and can execute them following a particular strategy. GATE provides a series of serial controllers (i.e. controllers that run their PRs in sequence):

`gate.creole.SerialController`: a serial controller that takes any kind of PRs.

`gate.creole.SerialAnalyserController`: a serial controller that only accepts Language Analysers as member PRs.

`gate.creole.ConditionalSerialController`: a serial controller that accepts all types of PRs and that allows the inclusion or exclusion of member PRs from the execution chain according to certain run-time conditions (currently features on the document being processed are used).

`gate.creole.ConditionalSerialAnalyserController`: a serial controller that only accepts Language Analysers and that allows the conditional run of member PRs.

e.g. *The following example shows how to create an ANNIE application and how to run it over a corpus*

```
// load the ANNIE plugin
Gate.getCreoleRegister().registerDirectories(new File(
 Gate.getPluginsHome(), "ANNIE").toURI().toURL());

// create a serial analyser controller to run ANNIE with
SerialAnalyserController annieController =
 (SerialAnalyserController) Factory.createResource(
     "gate.creole.SerialAnalyserController",
     Factory.newFeatureMap(),
     Factory.newFeatureMap(), "ANNIE");

// load each PR as defined in ANNIEConstants
for(int i = 0; i < ANNIEConstants.PR_NAMES.length; i++) {
  // use default parameters
  FeatureMap params = Factory.newFeatureMap();
  ProcessingResource pr = (ProcessingResource)
      Factory.createResource(ANNIEConstants.PR_NAMES[i],
                             params);
  // add the PR to the pipeline controller
  annieController.add(pr);
} // for each ANNIE PR

// Tell ANNIE's controller about the corpus you want to run on
Corpus corpus = ...;
annieController.setCorpus(corpus);
// Run ANNIE
annieController.execute();
```

4.6 Persistent Applications

GATE allows the persistent storage of applications in a format based on Java serialisation. This is particularly useful for applications management and distribution. A developer can save the state of an application when he/she stops working on its design and continue developing it in a next session. When the application reaches maturity it can be deployed to the client site using the same method.

When an application (i.e. a *Controller*) is saved, GATE will actually only

save the values for the parameters used to create the Processing Resources that are contained in the application. When the application is reloaded, all the PRs will be re-created using the saved parameters.

Many PRs use external resources (files) to define their behaviour and, in most cases, these files are identified using URLs. During the saving process, all the URLs are converted relative URLs based on the location of the application file. This way, if the resources are packaged together with the application file, the entire application can be reliably moved to a different location.

API access to application saving and loading is provided by means of two static methods on the `gate.util.persistence.PersistenceManager` class:

`public static void saveObjectToFile(Object obj, File file):` saves the data needed to re-create the provided GATE object to the specified file. The Object provided can be any type of Language or Processing Resource or a Controller. The procedures may work for other types of objects as well (e.g. it supports most Collection types).

`public static Object loadObjectFromFile(File file):` parses the file specified (which needs to be a file created by the above method) and creates the necessary object(s) as specified by the data in the file. Returns the root of the object tree.

e.g. *The following example shows how to save and load a GATE application*

```
//Where to save the application?
File file = ...;
//What to save?
Controller theApplication = ...;

//save
gate.util.persistence.PersistenceManager.
          saveObjectToFile(theApplication, file);
//delete the application
Factory.deleteResource(theApplication);
theApplication = null;

[...]
//load the application back
theApplication = gate.util.persistence.PersistenceManager.
                 loadObjectFromFile(file);
```

Chapter 5

Using Java for JAPE

5.1 Introduction

The RHS of a JAPE rule can consist of any Java code. This is useful for a variety of things such as removing temporary annotations, percolating and manipulating features from previous annotations, adding features to the document, etc.

The first rule below shows a rule which matches a first person name, e.g. “Fred”, and adds a gender feature depending on the value of the minorType from the gazetteer list in which the name was found. We first get the bindings associated with the person label (i.e. the Lookup annotation). We then create a new annotation called “personAnn” which contains this annotation, and create a new FeatureMap to enable us to add features. Then we get the minorType features (and its value) from the personAnn annotation (in this case, the feature will be “gender” and the value will be “male”), and add this value to a new feature called “gender”. We create another feature ”rule” with value “FirstName”. Finally, we add all the features to a new annotation “FirstPerson” which attaches to the same nodes as the original “person” binding.

```
Rule: FirstName

 (
  {Lookup.majorType == person_first}
 ):person
 -->
 {
 gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
```

```
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();
features.put("gender", personAnn.getFeatures().get("minorType"));
features.put("rule", "FirstName");
outputAS.add(person.firstNode(), person.lastNode(), "FirstPerson",
features);
}
```

Note that inputAS and outputAS represent the input and output annotation set. Normally, these would be the same (by default when using ANNIE, these will be the "Default" annotation set). Since the user is at liberty to change the input and output annotation sets in the parameters of the JAPE transducer at runtime, it cannot be guaranteed that the input and output annotation sets will be the same, and therefore we must specify the annotation set to which we are referring.

The next rule (contained in a subsequent grammar phase) makes use of annotations produced by the first rule described above. Instead of percolating the minorType from the annotation produced by the gazetteer lookup, this time it percolates the feature from the annotation produced by the previous grammar rule. So here it gets the "gender" feature value from the "FirstPerson" annotation, and adds it to a new feature (again called "gender" for convenience), which is added to the new annotation (in outputAS) "TempPerson". At the end of this rule, the existing input annotations (from inputAS) are removed because they are no longer needed. Note that in the previous rule, the existing annotations were not removed, because it is possible they might be needed later on in another grammar phase.

```
Rule: GazPersonFirst
(
 {FirstPerson}
)
:person
-->
{
gate.AnnotationSet person = (gate.AnnotationSet)bindings.get("person");
gate.Annotation personAnn = (gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();

features.put("gender", personAnn.getFeatures().get("gender"));
features.put("rule", "GazPersonFirst");
outputAS.add(person.firstNode(), person.lastNode(), "TempPerson",
features);
```

```
 inputAS.removeAll(person);
 }
```

5.1.1 A more complex example

The example below is more complicated, because both the title and the first name (if present) may have a gender feature. There is a possibility of conflict since some first names are ambiguous, or women are given male names (e.g. Charlie). Some titles are also ambiguous, such as "Dr", in which case they are not marked with a gender feature. We therefore take the gender of the title in preference to the gender of the first name, if it is present. So, on the RHS, we first look for the gender of the title by getting all Title annotations which have a gender feature attached. If a gender feature is present, we add the value of this feature to a new gender feature on the Person annotation we are going to create. If no gender feature is present, we look for the gender of the first name by getting all firstPerson annotations which have a gender feature attached, and adding the value of this feature to a new gender feature on the Person annotation we are going to create. If there is no firstPerson annotation and the title has no gender information, then we simply create the Person annotation with no gender feature.

```
Rule: PersonTitle
Priority: 35
/* allows Mr. Jones, Mr Fred Jones etc. */

(
 (TITLE)
(FIRSTNAME | FIRSTNAMEAMBIG | INITIALS2)*
 (PREFIX)?
 {Upper}
 ({Upper})?
 (PERSONENDING)?
)
:person -->
 {
gate.FeatureMap features = Factory.newFeatureMap();
gate.AnnotationSet personSet = (gate.AnnotationSet)bindings.get("person");

// get all Title annotations that have a gender feature
 HashSet fNames = new HashSet();
    fNames.add("gender");
    gate.AnnotationSet personTitle = personSet.get("Title", fNames);
```

```
// if the gender feature exists
 if (personTitle != null && personTitle.size()>0)
{
  gate.Annotation personAnn = (gate.Annotation)personTitle.iterator().next();
  features.put("gender", personAnn.getFeatures().get("gender"));
}
else
{
  // get all firstPerson annotations that have a gender feature
    gate.AnnotationSet firstPerson = personSet.get("FirstPerson", fNames);

  if (firstPerson != null && firstPerson.size()>0)
  // create a new gender feature and add the value from firstPerson
 {
  gate.Annotation personAnn = (gate.Annotation)firstPerson.iterator().next();
  features.put("gender", personAnn.getFeatures().get("gender"));
 }
}
  // create some other features
  features.put("kind", "personName");
  features.put("rule", "PersonTitle");
  // creat a Person annotation and add the features we've created
outputAS.add(personSet.firstNode(), personSet.lastNode(), "TempPerson",
features);
}
```

5.2 Adding a feature to the document

This is useful when using conditional controllers, where we only want to fire a particular resource under certain conditions. We first test the document to see whether it fulfils these conditions or not, and attach a feature to the document accordingly.

In the example below, we test whether the document contains an annotation of type "message". In emails, there is often an annotation of this type (produced by the document format analysis when the document is loaded in GATE). Note that annotations produced by document format analysis are placed automatically in the "Original markups" annotation set, so we must ensure that when running the processing resource containing this grammar that we specify the Original markups set as the input annotation set. It does not matter what we specify as the output annotation set, because the

annotation we produce is going to be attached to the document and not to an output annotation set. In the example, if an annotation of type “message” is found, we add the feature ”genre” with value ”email” to the document.

```
Rule: Email
Priority: 150

(
 {message}
)
-->
{
doc.getFeatures().put("genre", "email");
}
```

5.3 Finding the Tokens of a matched annotation

In this section we will demonstrate how by using Java on the right-hand side one can find all Token annotations that are covered by a matched annotation, e.g., a Person or an Organization. This is useful if one wants to transfer some information from the matched annotations to the tokens. For example, to add to the Tokens a feature indicating whether or not they are covered by a named entity annotation deduced by the rule-based system. This feature can then be given as a feature to a learning PR, e.g. the HMM. Similarly, one can add a feature to all tokens saying which rule in the rule based system did the match, the idea being that some rules might be more reliable than others. Finally, yet another useful feature might be the length of the coreference chain in which the matched entity is involved, if such exists.

The example below is one of the pre-processing JAPE grammars used by the HMM application. To inspect all JAPE grammars, see the muse/applications/hmm directory in the distribution.

```
Phase:  NEInfo

Input: Token Organization Location Person

Options: control = appelt

Rule:   NEInfo

Priority:100
```

```
({Organization} | {Person} | {Location}):entity
-->
{
  //get the annotation set
  gate.AnnotationSet annSet = ((gate.AnnotationSet)bindings.get("entity"));

  //get the only annotation from the set
  gate.Annotation entityAnn = (gate.Annotation)annSet.iterator().next();

  gate.AnnotationSet tokenAS = inputAS.get("Token",
        entityAnn.getStartNode().getOffset(),
        entityAnn.getEndNode().getOffset());
  List tokens = new ArrayList(tokenAS);
  //if no tokens to match, do nothing
  if (tokens.isEmpty())
     return;
  Collections.sort(tokens, new gate.util.OffsetComparator());

  gate.Annotation curToken=null;
  for (int i=0; i < tokens.size(); i++) {
    curToken = (gate.Annotation) tokens.get(i);
    String ruleInfo = (String) entityAnn.getFeatures().get("rule1");
    String NMRuleInfo = (String) entityAnn.getFeatures().get("NMRule");
    if ( ruleInfo != null) {
      curToken.getFeatures().put("rule_NE_kind", entityAnn.getType());
      curToken.getFeatures().put("NE_rule_id", ruleInfo);
    }
    else if (NMRuleInfo != null) {
      curToken.getFeatures().put("rule_NE_kind", entityAnn.getType());
      curToken.getFeatures().put("NE_rule_id", "orthomatcher");
    }
    else {
      curToken.getFeatures().put("rule_NE_kind", "None");
      curToken.getFeatures().put("NE_rule_id", "None");
    }
    List matchesList = (List) entityAnn.getFeatures().get("matches");
    if (matchesList != null) {
      if (matchesList.size() == 2)
        curToken.getFeatures().put("coref_chain_length", "2");
      else if (matchesList.size() > 2 && matchesList.size() < 5)
        curToken.getFeatures().put("coref_chain_length", "3-4");
```

```
    else
        curToken.getFeatures().put("coref_chain_length", "5-more");
    }
    else
      curToken.getFeatures().put("coref_chain_length", "0");
  }//for

}

Rule:   TokenNEInfo
Priority:10
({Token}):entity
-->
{
  //get the annotation set
  gate.AnnotationSet annSet = ((gate.AnnotationSet)bindings.get("entity"));

  //get the only annotation from the set
  gate.Annotation entityAnn = (gate.Annotation)annSet.iterator().next();

  entityAnn.getFeatures().put("rule_NE_kind", "None");
  entityAnn.getFeatures().put("NE_rule_id", "None");
  entityAnn.getFeatures().put("coref_chain_length", "0");
}
```

Chapter 6

Tools for Annotation and Evaluation

6.1 Manual annotation

There are two methods of manually annotating data in GATE. Usually, it is quicker to annotate by using the system to produce the majority of annotations automatically, and then manually editing the annotations produced.

6.1.1 Simple method of adding annotations

The simple method is when only the annotation type is needed, with no features. Annotations can be added simply by selecting the text and clicking on the annotation type that is displayed in the list of types.

6.1.2 Complex method of adding annotations

The more complex way of annotating documents is via the annotation editor, which is invoked via the menu displayed on right mouse click (after selecting the text to be annotated). This method is used in two cases: (i) when no annotations of the desired type already exist. The first annotation of this type needs to be created via the editor. Subsequent annotations can then be created either via the simple point-and-click method described above. (ii) when we want to annotate with more detailed information by specifying annotation features. For example, Date annotations in GATE can be several kinds (time, date, and date-time).

6.1.3 Modifying and removing annotations

To modify or remove an annotation, select the annotation from the annotations table, right click and select "Edit" or "Delete selected annotations". To delete an entire annotation type, select the annotation type from the annotation set on the right hand pane and use the keyboard Delete key. To delete an entire annotation set, select the annotation set in the right hand pane and use the keyboard Delete key. Note that the Default annotation set can be cleared but not removed.

6.2 Evaluation

There are two main tools for evaluation:

- Annotation Diff, which enables comparison of key and system annotations on a document;
- Corpus Benchmark Tool, which enables comparison of annotations on a whole corpus and over time.

Both tools give figures for precision, recall, F-measure and false positives.

6.2.1 Annotation Diff

Run the Annotation Diff tool by selecting it from the Tools menu. Select the key and response documents to be used (note that both must be open in GATE), the annotation sets to be used for each, and the annotation type to be evaluated.

The tool automatically intersects all the annotation types from the selected key annotation set with all types from the response set. On a separate note, you can perform a diff on the same document, between two different annotation sets. One annotation set could contain the key type and another could contain the response one.

After the type has been selected, the user is required to decide how the features will be compared. Features are compared by by analyzing if features from the key set are contained in the response set. Both feature name and value must be identical to obtain a match.

There are three basic options to select:

1. Take all the features from the key set into consideration;

2. Take only the user selected ones;

3. Ignore all features.

If false positives are to be measured, select the annotation type (and relevant annotation set) to be used as the denominator (normally, Token or Sentence). The weight for the F-Measure can also be changed - by default it is set to 0.5 (i.e. to give precision and recall equal weight). Finally, click on "Evaluate" to display the results. Note that the window may need to be resized manually, by dragging the window edges or internal bars as appropriate).

In the main window, the key and response annotations will be displayed. They can be sorted by any category by clicking on the relevant column header. The key and response annotations will be aligned if their indices are identical, and are color coded according to the legend displayed.

6.2.2 Corpus Benchmark tool

To use the corpus benchmark tool, first make sure the properties of the tool have been set correctly (see below). Then select "Corpus Benchmark Tool" from the Options menu. There are 3 ways in which it can be run:

- **Default mode** compares the stored processed set with the current processed set and the human-annotated set. This will give information about how well the system is doing compared with a previous version.
- **Human marked against stored processing results** compares the stored processed set with the human-annotated set.
- **Human marked against current processing results** compares the current processed set with the human-annotated set.

Once the mode has been selected, choose the directory where the corpus is to be found. The corpus must have a directory structure consisting of "clean" and "marked" subdirectories. The clean directory should contain the raw texts; the marked directory shuold contain the human-annotated texts. Finally, select the application to be run on the corpus (for "default" and "human v current" modes).

If the tool is to be used in Default or Current mode, the corpus must first be processed with the current set of resources. This is done by selecting "Store corpus for future evaluation" from the Corpus Benchmark Tool. Select the corpus to be processed (from the top of the subdirectory structure, i.e. the directory containing the marked and stored subdirectories). If a

"processed" subdirectory exists, the results will be placed there; if not, one will be created.

Once the corpus has been processed, the tool can be run in Default or Current mode. The resulting HTML file will be output in the main GATE messages window. This can then be pasted into a text editor and viewed in an internet browser for easier viewing.

The tool can be used either in verbose or non-verbose mode, by selecting the verbose option from the menu. In verbose mode, any score below the user's pre-defined threshold (stored in corpus_tool.properties file) will show the relevant annotations for that entity type, thereby enabling the user to see where problems are occurring.

6.2.3 How to define the properties of the benchmark tool

The properties of the benchmark tool are defined in the file corpus_tool.properties, which should be located in the directory from which Gate is run (usually gate/build or gate/bin).

The following properties should be set:

- the threshold for the verbose mode (by default this is set to 0.5);
- the name of the annotation set containing the human-marked annotations (annotSetName);
- the name of the annotation set containing the system-generated annotations (outputSetName);
- the annotation types to be considered (annotTypes).