Índice general

Índice de cuadros

Índice de figuras

Introducción

Inicialmente el objetivo que me plantee como proyecto de tesis fue elaborar un estado del arte en Procesamiento de Lenguaje Natural (PLN) alrededor de la arquitectura GATE (General Architecture For Text Engineering)[8]. (El funcionamiento de GATE se describe en el capítulo 3, incluyendo una descripción de su instalación.)

Después de estudiar el funcionamiento de GATE así como el de sus módulos, en cierto momento me di cuenta que el POS Tagger de Hepple (uno de los módulos) no podía adaptarse para el español. Un inconveniente grande pues, como se verá en el capítulo 1 (Extracción de Información), el POS Tagging es la segunda subtarea en la tarea de extracción de información.

El módulo Hepple no es adaptable al español porque no es open source y por lo tanto me fue imposible cambiar el algoritmo de etiquetado inicial. El inconveniente fue que el etiquetado inicial en Hepple etiqueta a las palabras desconocidas con "NN" y "NNP". "NNP" si la palabra empieza con mayúscula y "NN" si no empieza con mayúscula, según el sistema Penn Treebank. Pero yo tenía que etiquetar según el sistema CLIC-TALP (el único corpus en español que tuve disponible), y en consecuencia era obligatorio que el Hepple me etiquetara en ese sistema.

Siguiendo una idea, acaso un tanto romántica, hacia el final de la licenciatura me interesé en la filosofía Open Source y las soluciones al problema de reuso en el desarrollo de software. (Este contexto teórico del reuso y open source lo desarrollo en el capítulo 2 sobre ingeniería de software.) Como se sabe, el nivel de reuso en la Ingeniería de Software ha sido muy bajo. Una de las razones es quizá que los programas tienden a ser dependientes de la plataforma y también porque no han sido pensados teniendo en mente su reusabilidad, lo cual crea problemas de integración por carecer de una interfaz adecuada. La arquitectura y el entorno de desarrollo de GATE sí están pensados para la integración de módulos orientados a la resolución de problemas de procesamiento de lenguaje natural. Ésta es la razón por la que decidí estudiar a GATE.

Desgraciadamente, a pesar de que GATE ha declarado públicamente que su arquitectura es Open Source, en particular el módulo Hepple no lo es. Y esto me creó una dificultad. Así pues, esta dificultad de no poder adaptar el Hepple para el español me impuso el subproblema de tener que buscar un módulo sustituto (o bien abandonar todo el proyecto de usar la arquitectura GATE para extracción de información en español). La pista para la solución de este subproblema la encontré en el mismo módulo Hepple, pues Hepple está basado en Brill (Hepple es un Brill-based POS Tagger).

Como se verá en el capítulo 4, Hepple utiliza dos de las cuatro listas que genera el módulo de entrenamiento del Brill POS Tagger: utiliza sólo el lexicón y las reglas de contexto. Como yo sabía desde que empecé a estudiar el Hepple que éste está basado en Brill, ya tenía instalado en mi computadora el Brill POS Tagger y lo había probado, e incluso ya lo había entrenado para el español (Ver capítulo 5).

La solución a esta dificultad de la imposibilidad de adaptar el Hepple para el español, fue desarrollar un módulo sustituto, también basado en Brill. Así que decidí programar el módulo sustituto de Hepple, utilizando la misma idea que observé en el módulo Hepple de GATE de encapsularlo en un wrapper para poder acoplarlo a GATE pero también para que me permitiera usarlo en forma independiente. Sin embargo, a diferencia de GATE que utilizó un wrapper para ocultar el código, yo usé un wrapper como interfaz que permite el acoplamiento con GATE. El resultado fue el VMP Tagger que se describe en el capítulo 5.

En resumen, mi aportación en esta tesis consiste en 1) la utilización del Brill POS Tagger para entrenamiento para el español con lo cual se salva un obstáculo para el procesamiento de lenguaje natural de textos en español utilizando GATE; 2) el desarrollo del módulo VMP Tagger que sustituye al módulo de GATE (Hepple) que es de código cerrado y en consecuencia no adaptable para el español. Estas dos aportaciones permiten ya ejecutar la tarea de extracción de información para el español después de adaptar otros módulos lo cual es un proyecto que continuaré en mi tesis de maestría.

Además del etiquetador morfosintáctico presentado aquí y que sustituye al módulo Hepple de GATE, se necesita adaptar al español otros módulos de GATE como el módulo de categorización de sustantivos, el analizador sintáctico de superficie y el de resolución de correferencia. La adaptación de estos módulos de GATE para el español es una tarea futura.

A pesar del obstáculo al objetivo inicial que representó la imposibilidad de adaptar el etiquetador de Hepple y que me obligó a desarrollar todo un módulo acoplable a la arquitectura GATE, el objetivo inicial se logró en dos sentidos: 1) conocer de cerca el problema general del Procesamiento de

Lenguaje Natural, 2) conocer y operar la arquitectura GATE y adicionalmente 3) logré elaborar un inventario parcial del tipo de recursos que están actualmente disponibles en ese campo. El VMP Tagger es una pequeña contribución a él.

Capítulo 1

Extracción de Información

El objetivo de un sistema de extracción de información es obtener información acerca de hechos muy específicos (como fechas, nombre propios, eventos, relaciones entre eventos y entidades) a partir de un texto en lenguaje natural[1] acerca de un dominio de interés. La información obtenida como output puede ser mostrada directamente a los usuarios, o bien puede ser almacenada en una base de datos.

Por ejemplo, los analistas financieros necesitan estar al tanto del movimiento de la Bolsa, de las cotizaciones accionarias de diferentes empresas, de acuerdos entre empresas, e incluso necesitan estudiar la situación política y económica mundial. Esta información se puede encontrar en los periódicos y en los servicios de news wire[2].

Por otro lado, los sistemas de recuperación de información (Information Retrieval) son usados para obtener los artículos relevantes respecto a un tema dentro de un volumen muy grande de textos. Es lo que hace Google y otras search engines como Altavista, Lycos, Yahoo, etc. Sin embargo, una vez que se tienen todos lo artículos relevantes, el analista puede necesitar identificar cierta información dentro de ellos. En ese momento es necesario usar un sistema de *extracción de información* para encontrar los hechos específicos de interés sin necesidad de que el analista lea cada artículo.

El proceso de extracción de información se puede realizar más fácilmente en clases de textos que tengan información muy específica, expresable en forma de tabla o plantilla. Por ejemplo, textos referentes al movimiento de

[1] Sin estructura formal identificable por la computadora. Un texto en lenguaje natural es "visto"por la computadora como una cadena de símbolos

[2] Es un servicio de cable que transmite noticias actualizadas al minuto, usualmente en forma electrónica y a los medios masivos de comunicación. Para un ejemplo de un newswire comercial ver PR Newswire en http://www.prnewswire.com/

la Bolsa, desastres naturales, atentados terroristas, etc. (Al tema principal de estos textos suele denominársele dominio.) En estos dominios hay puntos muy claros que son relevantes: la empresa X ganó Y puntos el día Z. (En el caso de un atentado terrorista es de importancia saber el número de heridos, muertos, fecha, lugar, etc.)

La plantilla comúnmente se diseña tomando en cuenta las siguientes categorías:

- entidades: personas, organizaciones, lugares, fechas, etc.

- atributos (de las entidades): como título de una persona, tipo de organización, etc.

- relaciones (que existen entre las entidades): la organización X está ubicada en el país Y

- eventos (en los cuales las entidades participan): la empresa X firmó un acuerdo con la empresa Y

1.1. Tareas de la Extracción de Información

Las tareas de la extracción de información se caracterizan por dos propiedades importantes[3]:

a) el nivel de conocimiento requerido puede ser descrito mediante plantillas relativamente simples con espacios que necesitan ser llenados con material del texto insumo y,

b) sólo una pequeña parte del texto es relevante para llenar los espacios de la plantilla, el resto puede ser ignorado.

Se han creado varios proyectos que tienen como objetivo estimular el desarrollo de nuevos sistemas de extracción de información y crear un estándar para evaluar su desempeño. Los dos más importantes son los llamados Message Understanding Conference (MUC[4]) y Automatic Content Extraction (ACE[5]). La MUC tuvo lugar cada dos años durante el periodo 1987-1998, ACE está vigente desde principios de los noventa.

[3] http://www.ai.sri.com/∼appelt/ie-tutorial/
[4] http://www.itl.nist.gov/iad/894.02/related_projects/muc/index.html
[5] http://www.nist.gov/speech/tests/ace/

Las tareas de los sistemas de EI pueden diferir pues dependen de cómo se quiera modelar el sistema y del dominio específico en el que se esté trabajando. A continuación se presenta la clasificación de tareas utilizada en la última conferencia MUC (MUC-7).

1.1.1. Reconocimiento de Nombres (Named Entity Recognition)

En esta tarea se identifican los nombres propios (entidades, en la terminología del PLN) y se clasifican. Es decir, no sólo hay que reconocer si una palabra es un nombre propio, sino que hay que identificar si esa palabra se refiere a una persona, a un lugar, a una organización, etc.

1.1.2. Construcción de Plantilla de Elementos (Template Element construction)

En esta tarea, a cada entidad se le asocia un conjunto de atributos que la describen. Es decir, añade a las entidades información descriptiva usando correferencia. Tal asociación, lo mismo que la definición de los atributos descriptores, es dependiente del dominio. Por ejemplo, a una empresa conviene asociarle información acerca de su giro, valor, localización, etc.; a una persona puede asociársele su título (Sr., Lic., Ing.,), puesto en la empresa, etc. Los atributos de una entidad extraídos en esta etapa se agrupan en su plantilla correspondiente. En ésta, cada entidad tiene una id (identificación), e.g. ENTITY-2 la cual puede ser usada para hacer referencias cruzadas (cross-referencing) entre entidades y para describir eventos que involucran entidades. Cada entidad tiene también un tipo o categoría, e.g. compañía, persona.

1.1.3. Construcción de Plantilla de Relaciones (Template Relation construction)

El objetivo de esta tarea es encontrar las relaciones entre las plantillas de elementos obtenidas en la tarea anterior. Aquí también las relaciones son dependientes del dominio; por ejemplo, las personas y empresas pueden relacionarse mediante la relación "empleado de", las empresas y lugares se podrían relacionar mediante la relación "localizada en". A continuación se presenta un ejemplo de dos plantillas de entidades que incluyen una relación.

```
IBM
id entidad-2
```

```
tipo compañía
giro computadoras
localizada_en entidad 13

Estados Unidos
id entidad-13
tipo lugar
subtipo país
```

1.1.4. Construcción de Plantilla de Escenario (Scenario Template construction)

La plantilla de escenario es el resultado típico de un sistema de extracción de información. Esta tarea se encarga de relacionar las plantillas de elementos con los eventos de interés, por ejemplo, se puede relacionar a dos organizaciones A y B por medio del evento "creación de empresa conjunta (joint venture)".

```
creación de empresa conjunta
id evento-2
giro servidores
compañías entidad-2 entidad-4
```

1.2. Arquitectura de un Sistema de Extracción de Información

Según Appelt and Israel (1999), los dos enfoques básicos para la construcción de sistemas de extracción de información son el enfoque de ingeniería del conocimiento y el de métodos empíricos (sistemas entrenados automáticamente):

- En el enfoque de ingeniería del conocimiento es necesario el uso de expertos para analizar un *corpus* y construir gramáticas a partir de él. El corpus consiste en un conjunto de textos representativos de los que se desea extraer información. Las gramáticas son extraídas de ese corpus descubriendo en él patrones estructurales.

- En los sistemas entrenados automáticamente se utilizan, en cambio, métodos estadísticos y algoritmos que puedan generar reglas a partir de un *corpus anotado* manualmente y "legible"para un sistema de PLN.

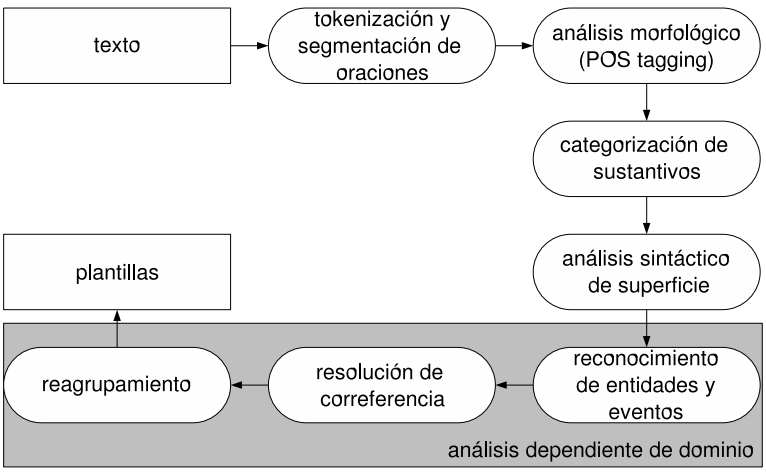

Figura 1.1: Arquitectura típica de un sistema de Extracción de Información

> El corpus anotado puede ser creado exprofeso, o bien puede usarse uno de los ya existentes creados por grupos de investigación.

Claramente nada impide que existan sistemas híbridos. Es el caso del *Brill tagger*, el cual usa un enfoque de entrenamiento automático pero da la opción de mejorar éste mediante el uso de la inspección experta.

Para poder realizar las tareas de la extracción de información que fueron descritas en el apartado anterior, el texto necesita ser "preprocesado" (con algunos módulos que le den una estructura adecuada) para entrar como insumo a un sistema de PLN. Por ejemplo, se necesita saber dónde se termina una oración y comienza otra, o bien, si una palabra es un verbo o un sustantivo o pronombre etc. Y ya sabiendo que es sustantivo, es necesario saber si es femenino o masculino, singular o plural.

Entonces, aunque la forma de modelar un sistema de extracción de información puede diferir dependiendo del dominio, hay una estructura básica que se debe tomar en cuenta independientemente del enfoque que se elija (experto o estadístico). La figura 1.1 presenta un diagrama de bloques de la arquitectura típica de un sistema de extracción de información. También presentamos a continuación un texto dentro del dominio muy específico de empresas conjuntas (joint venture) que será tomado como referencia en la discusión siguiente.

> IBM firma un acuerdo con Great Wall para fabricar servidores en China

> AFP/Pekín.
> El gigante estadounidense de la informática IBM ha firmado un acuerdo con Great Wall Computer Shenzhen para crear una empresa conjunta de servidores destinados al mercado asiático, indicó este lunes la empresa china en su página de Internet.
> Propiedad de la empresa estadounidense en un 80 %, International Systems Technology Company (ISTC) estará en Shenzhen (sur de China) y producirá e-servidores de la serie x y p de International Business Machines (IBM), según un comunicado. El volumen de negocios que se marcan como objetivo para 2005 es de mil millones de dólares (753 millones de euros). Los detalles financieros del acuerdo no fueron revelados.

1.2.1. Tokenización y Segmentación de Oraciones (Tokenizer and Sentence Splitter)

Para que un texto en lenguaje natural pueda ser procesado por un sistema de PLN es necesario segmentar la cadena (string) de caracteres del texto en palabras y oraciones. Para ello es necesario marcar las fronteras de las palabras (es la tarea del tokenizador) y de las oraciones (tarea que efectúa el segmentador de oraciones). Esta tarea es relativamente independiente del dominio.

1.2.2. Análisis Morfosintáctico (Part Of Speech Tagging)

Consiste en determinar la forma, clase o categoría gramatical de cada palabra de una oración. No debe confundirse ni mezclarse con el *análisis sintáctico*. La tarea del POS tagger es un poco más dependiente del dominio, pues depende del corpus, particularmente de los nombres propios. Sin embargo, aquí no se requiere el reconocimiento de objetos y eventos específicos del dominio. Una de las dificultades del análisis morfosintáctico es la desambiguación. Por ejemplo, la palabra "café"puede ser sustantivo o adjetivo y el pos tagging debe resolver estas ambigüedades tanto como sea posible. Ver capítulo 4 para un análisis detallado de esta tarea.

1.2.3. Categorización de Sustantivos

Un etiquetador morfológico llega a determinar si una palabra es nombre propio, pero no está dentro de sus funciones el determinar si es un nombre de persona, el apellido, o el nombre de una compañía. Es por eso que en esta etapa se hace uso de un lexicón para clasificar estos nombres propios.

Los lugares geográficos, las fechas, nombres de personas y de compañías son, con mucha frecuencia, palabras complejas (que consisten de más de una palabra, como "Río Bravo"). Entonces es necesario agrupar palabras para formar palabras complejas de tal manera que se puedan identificar como una sola unidad léxica. Para resolver este problema es conveniente contar con un lexicón para cada categoría de palabras. Por ejemplo, una lista de palabras de nombres propios, una de compañías, una de países, una de monedas, etc.

En un primer análisis el programa podría no identificar algunos nombres propios por ser palabras complejas y no estar en ningún lexicón. Por ejemplo, una empresa recién creada denominada International Systems Technology Co. En esos casos se puede requerir un análisis subsecuente. Por ejemplo, utilizando ciertas reglas de inferencia como "si las palabras empiezan con mayúscula y son seguidas por S.A. o Co., entonces probablemente se trata del nombre de una empresa".

1.2.4. Análisis Sintáctico de Superficie (Shallow Parsing)

En este nivel, las oraciones se segmentan en sintagmas nominales, sintagmas verbales y partículas[6]. Este análisis es efectuado conforme a una gramática, es decir, una serie de reglas que dicen qué combinaciones de categorías gramaticales forman una frase bien formada (gramaticalmente correcta).

A continuación se presenta el análisis sintáctico del primer párrafo del texto-ejemplo en el dominio de empresas conjuntas. NOTA: SN (Sintagma Nominal), SV (Sintagma Verbal), P (Preposición).

SN El gigante estadounidense
P de
SN la informática
SN IBM
SV ha firmado
SN un acuerdo
P con
SN Great Wall Computer Shenzhen
P para
V crear
SN una empresa conjunta
P de SN servidores

[6]Las partículas son todas las palabras que no son verbos ni sustantivos, como artículos, preposiciones, conjunciones y adverbios

SV destinados
P al (a)
SN (el) mercado asiático,
SV indicó
SN este lunes
SN la empresa china
P en
SN su página
P de
SN Internet.

1.2.5. Fases Dependientes del Dominio

Una vez que se han reconocido los sintagmas nominales y verbales, el sistema continúa con la identificación de entidades y eventos. Las entidades se identifican a partir de los sintagmas nominales, y los eventos a partir de los sintagmas verbales.

Reconocimiento de Entidades y Eventos

Del conjunto de sintagmas nominales hay que reconocer las entidades que se refieran a compañías, lugares geográficos, etc. Los demás sintagmas nominales se utilizarán como atributos de entidades y otros valores requeridos en la plantilla que ha sido ya diseñada y que requiere ser llenada. Lo mismo puede ser dicho de los eventos asociados a los sintagmas verbales.

Resolución de Correferencia

En todas las etapas anteriores el análisis se estaba haciendo a nivel oración. En esta fase, se operará a nivel texto para identificar las relaciones entre entidades. La resolución de correferencia es el proceso que se encarga de identificar múltiples representaciones de una misma entidad. En nuestro ejemplo, *la empresa china* se refiere a *Great Wall*

Reagrupamiento

En esta etapa se combinan las tareas de extracción de información descritas en la etapa anterior para el llenado de plantillas. Es decir, la información obtenida en la fase de reconocimiento de entidades y eventos y en la de resolución de correferencia es usada para el llenado de plantillas.

Capítulo 2

Ingeniería de Software: Calidad, Reuso y Modelos de Desarrollo

2.1. El Problema del Reuso de Software

Cuando se reutilizan componentes durante el desarrollo de software, se invierte menos tiempo en la creación de planes, modelos, documentos, código. El cliente recibe un producto con el mismo nivel de funcionalidad pero con menos esfuerzo de desarrollo. Así, el reuso mejora la productividad.

Un componente de software desarrollado para ser reusado estará libre de defectos. Pues, cuando se reusa y se encuentran defectos, éstos son eliminados. Con el paso del tiempo, el componente se vuelve virtualmente libre de defectos. Así, el reuso asegura la calidad. Pero el reuso favorece también una reducción en el costo, pues se invierte menos tiempo en el desarrollo. (Más sobre calidad y eficiencia en el desarrollo en la siguiente sección).

Sin embargo, entre los investigadores del área de Procesamiento de Lenguaje Natural la cantidad de recursos reutilizados es muy baja. La mayor parte de los recursos lingüísticos que se reutilizan están más hacia el lado de los datos (e.g. corpus, lexicones) que hacia el de los algoritmos.

Existen muchas barreras que impiden que se reutilicen componentes. A continuación se enlistan algunas:

- Dificultad para integrar componentes ya existentes. Por ejemplo, porque los desarrolladores usan diferentes lenguajes de programación o distintos paradigmas. O bien, porque la mayoría de las veces los desarrolladores no se esfuerzan lo suficiente en diseñar o definir la interfaz

del programa de aplicación (application program interface, API) ni documentan.

- Los desarrolladores no conocen los componentes existentes.
- Los desarrolladores no confían en la calidad del componente.
- Los componentes existentes son dependientes de la plataforma y son difíciles de instalar.
- Problemas con las políticas de licencias, de patentes o de costos.

Una forma de fomentar el reuso de software es a través de frameworks y arquitecturas. El framework de aplicaciones es un diseño reusable construido a partir de un conjunto de clases, y un modelo de colaboración entre los objetos. Un framework de aplicaciones es el esqueleto de un conjunto de aplicaciones que pueden ser adaptadas para su uso por un desarrollador de software. Un framework proporciona un conjunto de clases que cuando son instanciadas trabajan conjuntamente para lograr ciertas tareas en el dominio de interés. Uno de los aspectos distintivos más importantes de un framework son sus puntos de acoplamiento, es decir, los lugares del framework donde el diseñador puede introducir las variantes o diferencias para una aplicación particular.

Las definiciones de framework y arquitectura según el grupo de investigadores de GATE (ver capítulo 3) son las siguientes:

- Framework: Comúnmente, se usa framework para referirse a una biblioteca (library) de clases orientadas a objetos que se ha diseñado teniendo cierto dominio de aplicación en mente y que pueden ser adaptadas y extendidas para resolver problemas de ese dominio. (Otros autores le llaman a esto toolkit o library.) Los frameworks también son conocidos como plataformas o sistemas de componentes.
- Arquitectura: La arquitectura de software se ocupa de la estructura del sistema, de la organización del software, de la asignación de responsabilidades a los componentes y de asegurar que las interacciones entre los componentes satisfagan los requerimientos del sistema.

En la figura 2.1 se presenta un infograma que ayuda a distinguir entre framework y arquitectura. La figura de la izquierda, donde el área sombreada es la parte de reuso del sistema, representa un framework. La de la derecha, representa una arquitectura. Es importante hacer notar desde ahora que GATE es, a la vez, un framework y una arquitectura. Es decir, al tiempo

que proporciona una estructura y una organización, también proporciona una biblioteca de clases listas para ser adaptadas y extendidas para resolver problemas del procesamiento de lenguaje natural. El funcionamiento de GATE se aborda en el capítulo 3.

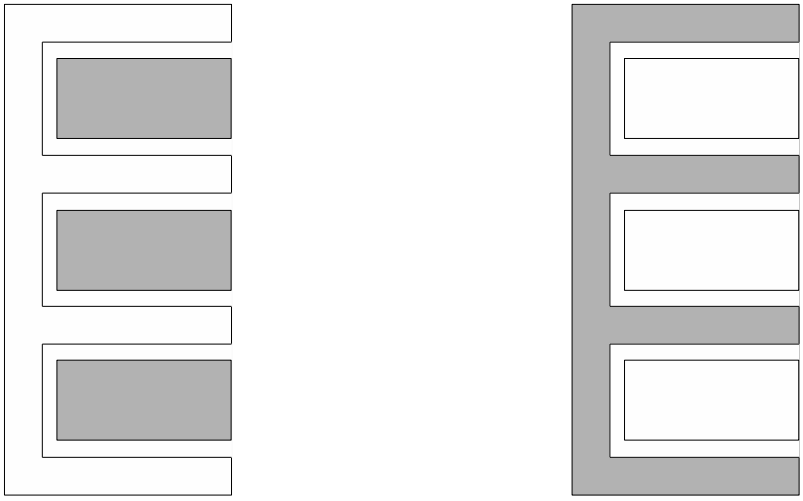

Figura 2.1: Arquitectura y framework

2.1.1. La Tecnología Java Beans y el Enfoque de Componentes de Software

Uno de los primeros esfuerzos orientados a resolver el problema del reuso del software son las metodologías y la programación orientada a objetos. Su concepto y herramienta central son las clases.

Un objeto es un grupo de variables (atributos), y está equipado con un grupo de operaciones que tienen acceso a esas variables y operan sobre ellas. Una clase es un conjunto de objetos similares. Todos los objetos de una clase dada tienen las mismas variables y están equipados con las mismas operaciones. En la terminología orientada a objetos las variables de un objeto son llamadas también variables de instanciación (instance variables) o variables de membresía (member variables), y las operaciones de un objeto usualmente reciben el nombre de constructores y métodos.

Un constructor es una operación que crea (e inicializa) un nuevo objeto de la clase. En C++ y Java, un constructor siempre recibe el nombre de la clase a la que pertenece. Un método es una operación que inspecciona y/o actualiza un objeto existente de una clase [28].

Sin embargo, las expectativas de reuso de las clases no se cumplieron. Porque en C++ las clases sólo se pueden usar dentro del contexto específico de las aplicaciones para las que fueron desarrolladas y son dependientes de la plataforma. Y en el caso de Java las clases son funcionalmente similares a las de C++ aunque añaden el beneficio de ser multiplataforma. De cualquier manera la programación orientada a objetos dio un gran paso en la dirección correcta para resolver el problema del reuso. Porque las clases hacen posible que los programadores se olviden de procedimientos y datos y manejen solamente objetos.

Pero la aspiración de los desarrolladores de software en la era de Internet es que el código pueda ser reusado independientemente de plataforma y del lenguaje de programación y darle así al software una oportunidad para crecer y expandirse. La solución a esta aspiración parece haberla proporcionado la tecnología de componentes de software desarrollada por Sun Microsystems [20]. Se trata de los Java Beans. Esta tecnología de componentes denominada JAVA Beans es una API (Application Program Interface) independiente de plataforma y de arquitectura. Permite crear y usar componentes Java de software[1].

Una forma fácil de entender el software de componentes es con ejemplos. Un subconjunto de componentes está constituido por las componentes visuales que son componentes de software que tienen una representación visual. El ejemplo más simple de un componente visual es un botón (button), el cual es un elemento gráfico que puede ser usado por muchas herramientas builder (Application Builder Tools). Un botón funciona como una unidad autocontenida fácilmente integrable en aplicaciones. Y con un builder es muy fácil de añadir: basta hacer un click.

Puesto que GATE está basado en Beans, necesitamos algunas definiciones más sobre los Java Beans. La arquitectura definida por un modelo de componentes de software se encarga de determinar cómo van a interactuar las componentes en un entorno dinámico. El software de componentes define dos elementos fundamentales: componentes y contenedores. La parte de componentes se encarga de definir las formas en que las componentes

[1]A pesar de que la documentación de Java habla de componentes, otro término equivalente y usado también con mucha frecuencia es el de módulos. La idea básica es el ensamblado de unidades autocontenidas de software (módulos o componentes) para formar aplicaciones más complejas. Y a esta idea básica se le llama modularidad. Es decir, un sistema que consiste de módulos acoplados se dice que es modular. Sería más difícil de aceptar la palabra componencialidad. Por eso, a pesar de que aquí estamos usando la documentación de Java y por eso usamos componentes, en otros lugares usaremos módulos en forma equivalente

se crean y usan. Es decir, proporciona la plantilla (template) a partir de la cual van a ser creadas componentes concretas. La parte de contenedores (del software de componentes) define un método para combinar componentes y formar con ellas estructuras útiles. Los contenedores proporcionan el contexto para la organización y la interacción de las componentes.

Además de definir la estructura de componentes y contenedores, un modelo de software de componentes también es responsable de proporcionar servicios. Los principales son:

- Introspección
- Manejo de Eventos
- Persistencia
- Layout
- Soporte de una herramienta builder
- Soporte para computación distribuida

La introspección es el mecanismo que permite que la funcionalidad de los componentes puedan mostrarse. Una aplicación puede interrogar a un componente acerca de sus capacidades, determinando así las formas de interacción. En GATE, cada componente de software incluye un archivo en XML en un formato especial de GATE que describe sus características.

El manejo de eventos es el mecanismo que hace posible que un componente genere notificaciones correspondientes a cambios en su estado interno. Un evento es algo que sucede dentro de un componente y que una aplicación u otro componente puede necesitar saber para responder de una cierta manera. En el ejemplo del botón ya mencionado, éste genera un evento cuando es clickeado.

Persistencia es el servicio proporcionado por un modelo de componentes de software mediante el cual un componente es almacenado en y recuperado de un medio de almacenamiento no volátil tal como un disco duro. La información que se almacena y recupera es el estado interno del componente, así como su relación a un contenedor u otros componentes. En GATE se puede optar por Save Session on Exit o bien Save Options on Exit.

Típicamente el layout soportado por un sistema de componentes de software consiste en proporcionar a la componente un área rectangular en la cual pueda mostrarse visualmente y pueda ser manipulada en el contexto

de un contenedor que aloja a otros componentes. En las herramientas builder el desarrollador despliega las componentes al estar construyendo una aplicación.

El soporte de una herramienta builder posibilita a los usuarios el poder construir gráficamente aplicaciones complejas a partir de los componentes. Las herramientas de desarrollo usan las propiedades y conductas exhibidas por los componentes para adaptar e integrar componentes en el contexto de una aplicación. Con frecuencia la herramienta builder consiste de cuadros de diálogo que posibilitan que el usuario edite gráficamente las propiedades de un componente.

Finalmente el soporte de computación distribuida se ha quedado en una aspiración, por lo menos en GATE. El procesamiento distribuido permitiría la construcción de sistemas con componentes residentes en diferentes computadoras.

2.2. Calidad de Software: Métricas y Modelos

A pesar de que no hay un acuerdo en la literatura de la Ingeniería de Software sobre el significado de calidad de software, el tema es importante dado que la eficacia del desarrollo del software tiene que ser evaluada de alguna manera. A continuación se presenta una revisión de la literatura sobre calidad de software con la finalidad de construir por lo menos los parámetros y principios a partir de los cuales se puede hablar de calidad de software en forma relativamente clara.

Según Kan[22] la calidad del software puede ponderarse desde dos puntos de vista: el popular y el profesional. Desde el punto de vista popular la calidad se reconoce cuando se ve, no puede ser cuantificada. Pero desde el punto de vista profesional la calidad tiene que definirse y medirse. Kan define la calidad del software en dos niveles: calidad intrínseca del producto y satisfacción del cliente. Desde la filosofía de la Calidad Total, Kan asegura que para lograr la calidad del software se debe atender al éxito de largo plazo poniendo la satisfacción del cliente como elemento central.

Al abordar la cuestión de medición de la calidad, Kan introduce el concepto de métrica de software. Y clasifica tales métricas en tres categorías: del producto, del proceso, y del proyecto. La métrica del producto se ocupa de medir las características tales como tamaño, complejidad, diseño, desempeño, etc. La métrica del proceso incluye la eficacia de la eliminación de defectos (debugging) durante el desarrollo y se usa principalmente para mejorar el desarrollo y el mantenimiento del software. Finalmente, la métrica

del proyecto incluye el número de desarrolladores, los costos, la calendarización y la productividad; en general, describe las características del proyecto y de su ejecución. En cuanto a las métricas de calidad de software, Kan las clasifica en métricas de calidad del producto terminado y métricas de calidad del proceso. Y es la relación entre estas dos métricas lo que constituye la esencia de la ingeniería de calidad de software.

El proceso de desarrollo de software conduce a un producto terminado que se libera a un mercado de usuarios. Y es en esta interrelación entre proceso, producto y usuarios en que la mayoría de los autores en calidad de software centran su atención. Pero al existir diversas metodologías para el desarrollo de software, cada una con sus pretensiones de calidad, es evidente la necesidad de evaluar el proceso y también el producto (lo cual puede incluir sondeos sobre satisfacción de los usuarios) para así poder discriminar entre las metodologías diversas en el momento de elegir una. Pero, según Gentleman[12], la escasa evidencia experimental impide una evaluación informada de las pretensiones de calidad de aquéllas.

Gentleman sostiene que la calidad, al igual que la belleza, radica en el ojo del que la ve. Es decir, no es absoluta sino que depende de la perspectiva del evaluador. Y, sin embargo, cuantificar la calidad es importante para la toma de decisiones. Y esto a pesar de que la tarea se dificulta pues la calidad es multifacética: sus caras cambian con el contexto, incluso para la misma persona en diferentes momentos.

El estándar ISO/IEC 9126[19], que considera sólo atributos de significancia directa para el usuario, sugiere que hay varios puntos de vista potenciales para evaluar la calidad, entre los cuales pueden contarse el punto de vista del usuario, el de los desarrolladores, y el de los administradores. El enfoque directo para medir la calidad es estudiar las percepciones que otros se han formado acerca del software y extrapolarlas a nuestras situaciones. Tales otros deberían incluir expertos en el área, articulistas revisores de software, etc. Este enfoque es cualitativo, no es todavía una métrica cuantitativa.

Pero hay un enfoque indirecto que relaciona los atributos medidos con la medida de satisfacción del usuario. Este enfoque parece tener el atractivo de ser más cuantitativo y preciso que la presentación discursiva con listas de control, típica del enfoque directo. Y un enfoque todavía más indirecto consiste en estudiar el proceso mediante el cual se construyó el software. Este es el enfoque de la familia de estándares ISO 9000, que exige básicamente que, sin importar el proceso usado en el desarrollo, éste debería ser comprendido y documentado y debería ser posible monitorearlo para asegurar que esté realmente usándose.

Pero a pesar del tan publicitado mejoramiento de procesos, y de la na-

turaleza autoevidente de la afirmación de que el proceso debe hacer una diferencia, hay escasa evidencia experimental acerca de si las actuales metodologías tienen el efecto que dicen tener. En síntesis, la tesis de Gentleman es que la calidad del software es difícil de medir pero que tal tarea de medición es necesaria.

Los productos de software son sistemas complejos que, una vez construidos y puestos a funcionar, muestran sus defectos. Pero debido a la complejidad del programa, tales defectos son muy difíciles de eliminar. En opinión de Humphrey[17] esta es ya la actitud resignada y cómoda de la industria del software; consiste en ver los defectos del software como meras molestias, librándose así de la responsabilidad de corregirlos y de los daños que pudieran causar. Humphrey sostiene que tal actitud debe cambiar y propone una serie de reformas a la ley vigente.

La búsqueda y eliminación de defectos (debugging) en el software es una pesada tarea que debe ser ejecutada como una responsabilidad de la empresa que vende el producto y en favor del usuario final. Y en esta tarea de depuración se ha mostrado muy superior un modelo alternativo de desarrollo de software: el open source.

A partir de la década del 90 del siglo pasado empezó a ganar fuerza un modelo alternativo al tradicional de desarrollo de software que se incorporó al sistema legal norteamericano de la industria del software bajo la Open Source Definition [2]. La Open Source Definition incluye un conjunto de licencias para el software caracterizadas por dejar abierto el código de sus productos.

Raymond[25] sostiene que la experiencia de Linux sugiere la superioridad del modelo open source de desarrollo sobre le modelo tradicional centralizado. Esta tesis está basada en un experimento real que él mismo inició con un proyecto open source denominado Fetchmail. Otros autores han estudiado el modelo open source en búsqueda de la clave de su éxito.

Halloran y Scherlis[13], por ejemplo, examinaron proyectos exitosos de open source enfocándose a los aspectos técnicos de colaboración y calidad. Estos autores identificaron, en el muestreo que realizaron, algunas prácticas comunes del modelo open source de desarrollo. Una de ellas es el uso de herramientas open source para desarrollar proyectos open source, una práctica a la que denominan bootstrapping. También encontraron que, en los proyectos open source, cualquiera puede aportar pero no todos pueden decidir; pues son los líderes del proyecto quienes controlan la composición, la configuración y el flujo de la información que entra y sale del servidor de

[2] http://www.opensource.org/docs/definition.php

su proyecto.

En un proyecto de desarrollo de software es preciso atender algunas variables críticas que, según un estudio realizado por Schmidt y Porter[27], el modelo open source permite resolver con mayor facilidad que los enfoques tradicionales de desarrollo de software. Entre otros problemas que el open source resuelve mejor que el enfoque tradicional, estos autores identifican 1) el mantenimiento a largo plazo y la evolución de los costos, 2) el garantizar niveles aceptables de calidad, 3) mantener la confianza y la buena voluntad de los usuarios finales, y 4) asegurar la coherencia y las propiedades de usabilidad del sistema del software.

Koch y Schneider[23], por otro lado, aplicando una metodología que ellos mismos desarrollaron, estudiaron el proyecto open source GNOME[3] y encontraron que, a pesar de que en un proyecto open source hay más gente involucrada que en los proyectos tradicionales, un "círculo interno" relativamente pequeño de programadores es responsable de la mayor parte del trabajo. Este hallazgo coincide con el encontrado por Halloran y Scherlis.

Por su parte Hatton[14] evaluó el proyecto Linux con la metodología desarrollada por Software Engineering Institute (SEI) denominada Capability Maturity Model (CMM)[4]. El modelo CMM se caracteriza por clasificar a los proyectos en 5 niveles (Inicial, Replicable, Definido, Gestionado y Optimizado), donde el nivel Inicial es el más bajo o de calidad mínima. La evaluación de Hatton ubica a Linux en el nivel 1 (Inicial), el más bajo en esa metodología de medición, aunque reconoce que el proceso Linux de desarrollo tiene áreas de gran fortaleza. Los resultados de este estudio sugieren que la carencia de un modelo formal de desarrollo no ha sido un obstáculo para la producción de un sistema excepcionalmente confiable y de muy rápido desarrollo. (Ver datos sobre calidad de Linux al final de este capítulo.)

2.2.1. Open Source: Un Modelo Alternativo de Desarrollo de Software

El principio guía para el desarrollo de software Open Source es que, al compartir el código fuente, los desarrolladores cooperan bajo un modelo de rigurosa revisión de pares (peer-review) y sacan ventaja de la depuración de errores en paralelo (muchos beta-testers trabajando simultáneamente en la

[3]El proyecto GNOME (GNU Network Object Model Environment) surge en agosto de 1997 como proyecto liderado por Miguel de Icaza para crear un entorno de escritorio completamente libre para sistemas operativos libres, en especial para GNU/Linux. Ver http://www.gnome.org/

[4]http://www.sei.cmu.edu/cmm/

tarea de debugging) lo cual conduce a la innovación y al avance rápido en el desarrollo y la evolución de los productos de software.

Aún cuando no existe ningún artículo donde se describa formalmente el modelo de OSS (Open Source Software), Raymond[25] describe a la comunidad open source y sus métodos de desarrollo de software. El título del libro (The Cathedral and The Bazaar) es una metáfora: la producción propietaria (o de patente) de software como la construcción cuidadosamente planeada de una catedral, y la producción del software open source como las interacciones caóticas de los participantes en un bazar del medio oriente [5]. Y aunque esta analogía parecería quizá extrema, apunta a la diferencia principal entre los dos tipos de creación de software: sofisticada administración central contra desarrolladores y usuarios débilmente acoplados en miles de proyectos independientes.

Más allá de las connotaciones que pudieran tener los conceptos de centralización y descentralización, habría que decir que administración central no necesariamente implica eficiencia, ni la descentralización ineficiencia. De hecho, Raymond confiesa que él mismo no estaba convencido de la eficiencia o efectividad del modelo del bazar. Para convencerse lo puso a prueba iniciando el desarrollo de un proyecto que llamó Fetchmail (inicialmente Pop-Client) bajo los principios de desarrollo de Linus Torvald tratando de entender por qué el mundo Linux no se había desintegrado en la confusión sino que parecía reforzarse continuamente a una velocidad difícilmente imaginable para los constructores de catedrales. El proyecto Fetchmail tuvo un éxito significante.

Según Raymond[25], los modelos de la catedral y del bazar se basan en dos estilos de desarrollo fundamentalmente diferentes y se derivan de suposiciones opuestas acerca de la naturaleza de la depuración (debugging). Raymond propone algunos aforismos que describen el modelo open source de desarrollo. Enseguida se presentan algunos de ellos:

- Release Early, Release Often. And listen to your costumers. (Liberar pronto, liberar con frecuencia). Y escucha a tus usuarios.

- Ley de Linus: Given enough eyeballs, all bugs are shallow (Dada una base suficientemente grande de beta-testers y co-desarrolladores, casi cada problema será caracterizado rápidamente y su solución será obvia para alguien.)

- Los buenos programadores saben qué escribir y los grandes saben

[5] bazar = mercado en persa

qué reescribir (y reusar)[6].

- El tratar a tus usuarios como co-desarrolladores es la ruta con menos obstáculos para el mejoramiento rápido del código y la depuración efectiva[7].
- Si tratas a tus beta-testers como si fueran tu recurso más valioso, ellos responderán y se convertirán en tu recurso más valioso.
- Cualquier herramienta debería ser útil en la forma esperada, pero una herramienta verdaderamente grandiosa se presta a usos inesperados.

Un proyecto típico de Open Source empieza 1) con una persona que tiene un cierto interés en un problema de software y tiene algunas buenas soluciones posibles e 2) indaga con algunos amigos y colegas acerca sus conocimientos sobre el tema. (Posiblemente algunos de ellos tienen problemas similares aunque ninguna solución.) Después de algún tiempo 3) todas las personas interesadas empiezan a intercambiar sus conocimientos sobre el problema y crean así un bosquejo del asunto de su interés. Si hay suficiente interés los miembros del grupo desearán gastar algunos recursos en encontrar una solución para el problema dado, y 4) se creará entonces un proyecto informal. (Quienes no estén suficientemente interesados abandonan el grupo asegurando así que los que se quedan tienen un interés real de participar.) Con el tiempo y suficiente esfuerzo 5) los miembros del proyecto logran un resultado presentable y pueden así 6) publicar su trabajo en la red en un lugar donde mucha gente sea capaz de accesarlo[8]. (Quizá lo anuncien en algunos lugares como mailing lists, newsgroups –a la manera de Torvald– o servicios de noticias en línea.) Después de su publicación en la red 7) otras personas reconocen en el proyecto algunas de sus preocupaciones y se interesan en una solución conveniente también. (Si el interés de estos nuevos interesados es suficientemente grande, revisarán los resultados del proyecto –por ejemplo, usándolo– y al mirarlo desde una perspectiva diferente podrían sugerir algunas mejoras y empezar a comunicarse con el proyecto y, en consecuencia, se unirán a él.) Si el proyecto es realmente interesante 8) crecerá y la

[6]Como lo hizo Linus: reusó el código y las ideas de un sistema operativo llamado Minix. Y aunque eventualmente todo el código de Minix fue desechado o completamente reescrito, aportó la estructura para lo que eventualmente llegó a convertirse en Linux.

[7]Raymond cree que la aportación más ingeniosa y de mayores consecuencias de Linux fue, no la construcción del kernel de Linux, sino la invención del modelo Linux de desarrollo, con este principio como central.

[8]o bien puede solicitar su inclusión en el servidor de sourceforge www.sourceforge.net (en particular ver mi proyecto open source VMP Tagger en http://vmptagger.sourceforge.net/)

retroalimentación de parte de los miembros ayudará a lograr una mejor comprensión del problema y estrategias posibles para resolverlo. A medida que crece 9) la nueva información y los nuevos recursos se integran al proceso de investigación y 10) el ciclo se cierra reiniciándose en el punto 5).

2.2.2. Análisis del Modelo Open Source

Enseguida se presenta un análisis del modelo open source de desarrollo con base en el esquema de Zachman (y el de Checkland). Según Feller y Fitzgerald[10], el esquema de Zachman fue diseñado para la arquitectura de sistemas de información (SI) y contiene las categorías básicas de qué, cómo y dónde, para clasificar las diferentes arquitecturas de SI; estas categorías se complementan por las categorías descriptoras añadidas quién, cuándo y por qué[9]. Checkland por su parte ha propuesto una técnica llamada CATWOE (client, actor, transformation, weltanschauung, owner, enviroment) para el diseño de soft systems[10]. Las categorías de cómo y cuándo de Zachman no aparecen en el esquema de Checkland explícitamente pero el qué se puede asociar con transformation, el por qué con el weltanschauung o visión del mundo, el cuándo y el dónde con el enviroment y el quién con las categorías de client, actor, owner.

Los esquemas anteriores pueden ser aplicados al análisis del modelo open source de desarrollo de la siguiente manera:

1. Los qué o transformación. (¿Qué es?) Un programa de open source está estrictamente definido por la licencia bajo la cual es distribuido. (¿Qué productos/proyectos?) Software de infraestructura y de propósito general orientado a satisfacer necesidades no satisfechas por los mercados dominantes de software.

2. Los por qué o visión del mundo.

(Motivaciones tecnológicas): Necesidades de a) código más robusto, b) ciclos de desarrollo más rápidos, c) estándares altos de calidad, d) confiabilidad y estabilidad, y e) plataformas y/o estándares más abiertos.

(Motivaciones económicas): Necesidad corporativa de costos y riesgos compartidos, y la redefinición de la industria de bienes y servicios.

(Motivaciones humanas): Raymond lo explica en Homesteading the noosphere [25] en términos de una gift culture en la cual los participantes compiten por prestigio regalando tiempo, energía y creatividad.

[9] Ver http://www.businessrulesgroup.org/BRWG_RFI/ZachmanBookRFIextract.pdf para un desarrollo más amplio del esquema.

[10] en la terminología de Checkland soft systems se refieren a los sistemas de distribución de información en contraste con los hard systems que distribuyen objetos materiales.

3. Los cuándo y los dónde, el medio ambiente.

(Dimensiones temporales del modelo OS): el modelo OS se caracteriza por el desarrollo y evolución rápidos del software y por la liberación frecuente e incremental (cada versión nueva es liberada).

(Dimensiones espaciales): El modelo open source se caracteriza por los equipos distribuidos de desarrolladores en el ciberespacio.

4. Los cómo: El corazón de la metodología del modelo open source es el desarrollo y depuración masivos en paralelo, lo cual involucra contribuciones de desarrolladores individuales débilmente centralizados que cooperan gratuitamente o mediante retribuciones modestas.

5. Los quienes o clientes, actores y propietarios.

(Desarrolladores): Los desarrolladores del modelo open source se han autonombrado tradicionalmente hackers –pero no crackers–, desarrolladores profesionales –no amateurs–, autoseleccionados y altamente motivados. Ver más sobre esto en [25].

(Distribuidores): Las compañías distribuidoras del open source patrocinan a los desarrolladores de acuerdo a un modelo que desplaza al tradicional y que está orientado al servicio al cliente, a la administración de la marca, y a la reputación, como parámetros críticos.

(Usuarios): Hasta ahora los usuarios del open source han sido principalmente usuarios expertos y adoptadores tempranos.

2.3. Open Source y su Éxito en la Industria del Software

La historia del open source podría fecharse el 25 de agosto de 1991 cuando Linus Torvald[11] puso un mensaje a la comunidad newsgroups: comp.os.minix invitándola a colaborar en su proyecto que después se convertiría en el sistema operativo Linux:

```
Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't
be big and professional like gnu) for 386(486) AT clones.
(...) I'd like to know what features most people would
want. Any suggestions are welcome, but I won't promise
I'll implement them :-) (...) I got some responses to
```

[11] Ver http://www.resonancepub.com/linus.htm para una lista de los primeros mensajes de Linus Torvald y http://www.xmission.com/∼comphope/history/unix.htm para una cronología del desarrollo de UNIX/LINUX y sus variantes.

```
this (most by mail, which I haven't saved), and I even
got a few mails asking to be beta-testers for linux.
```

En http://www.opensource.org/advocacy/faq.php el lector puede encontrar mayor información sobre el significado del término open source (el cual se hace más difuso para la gente a medida que se hace más popular). Un sound bite de esa información se halla resumido en:

> *Open source promueve la calidad y la confiabilidad del software apoyando la evolución rápida de código fuente a través de la revisión entre colegas (peer review). Para ser certificado por la OSI (Open Source Initiative) el software debe distribuirse bajo una licencia que garantiza el derecho a leer, distribuir, modificar y usar el software libremente.*

En http://www.opensource.org/docs/definition.php el lector puede leer la versión 1.9 de la Open Source Definition, la cual debería darle una comprensión más amplia del término open source tal y como es usado en relación con el software. La "Open Source Definition" es una carta de derechos para el usuario de computadoras. Define ciertos derechos que una licencia de software debe garantizarle para ser certificada como Open Source. Las licencias de los programas de open source software (OSS) le otorgan al usuario los siguientes derechos:

- Correr el programa para cualquier propósito
- Estudiar y modificar el programa
- Distribuir libremente el programa original o el modificado

En otras palabras, las licencias open source abandonan los derechos esenciales del creador original garantizados por la ley copyright dándole a los usuarios más derechos sobre el producto de software de los usuales a los que están acostumbrados.

El desarrollo de software open source ha generado mucho interés en los últimos años, especialmente después del éxito de Linux. Tal éxito abre una promesa e invita a estudiarlo más a fondo con miras a encontrar en su modelo de desarrollo principios que puedan usarse con ventaja en la ingeniería de software. Sobre todo ante el fracaso relativo de ésta y la "crisis del software" (un término acuñado en la década de 1960, cf [2]) es decir, tiempos de desarrollo del software demasiado largos que exceden el presupuesto y con

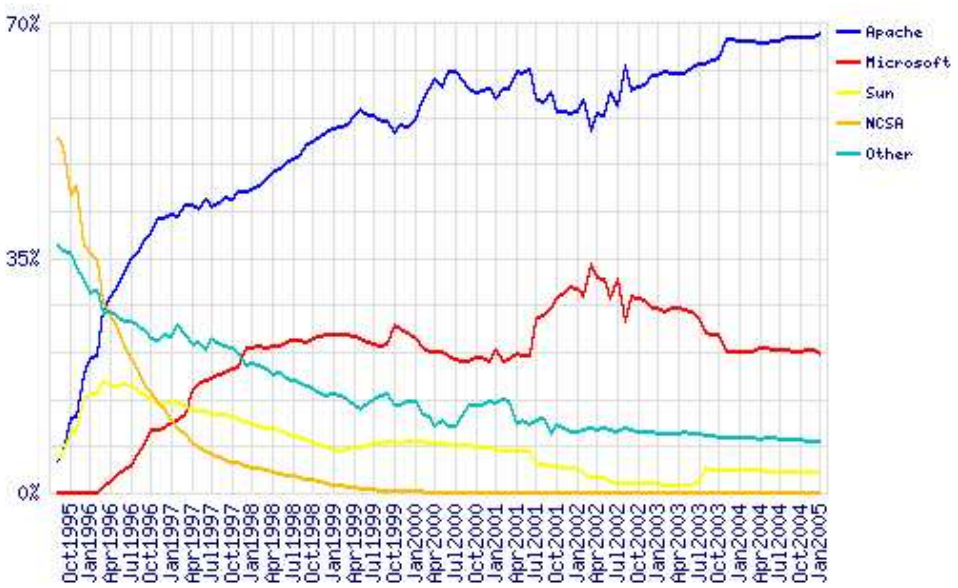

Figura 2.2: Principales servidores Web y su participación en el mercado sobre todos los dominios(Octubre 1995 - Agosto 2005)

productos que no funcionan muy bien. (Es dentro de esta perspectiva open source de desarrollo de software en busca de la eficiencia y la calidad que se puede ubicar esta tesis y su producto principal que es el VMP Tagger, un módulo acoplable a la arquitectura GATE.)

Además del sistema operativo Linux, hay otros productos open source exitosos[12]:

- Apache es actualmente el #1 de los servidores web (68.43 %). (Ver figura 2.2)
- PHP es el #1 de los lenguajes de script del lado del servidor
- Sendmail es el #1 de los servidores de email (Sendmail 42 %, Microsoft Exchange 18 %)

Como ya se dijo antes, la calidad del producto y el proceso de desarrollo de software están íntimamente relacionados. Cuando un producto es puesto

[12]Para ver más ejemplos el lector puede visitar http://www.dwheeler.com/oss_fs_why.html

a prueba y se encuentra que es de baja calidad, generalmente ello significa que el proceso de desarrollo no pudo ser completado dentro del tiempo programado y/o dentro del presupuesto. A la inversa, un proceso de baja calidad generalmente producirá un producto de baja calidad [18]. Es por esto que el estudio del modelo open source promete ser de gran valor para la ingeniería de software en vista de la alta calidad de sus productos como lo muestran los siguientes datos para el caso Linux[13].

- Confiabilidad: GNU/Linux es más confiable que Windows NT, según estudios de un año de Bloor Research:
  - GNU/Linux se cayó una vez
  - Windows NT se cayó 68 veces
- Desempeño: En el 2002, las medidas de la base de datos TPC-C (The Transaction Processing Performance Council http://www.tpc.org/) encontraron que Linux es más rápido que Windows 2000.
- Escalabilidad: OSS ha desarrollado sistemas de software a gran escala: Red Hat Linux 7.1 tiene 30 millones de líneas de código fuente (SLOC). Representa aproximadamente 8,000 persona-año o 1 billón de dólares.

[13]Para datos sobre otros proyectos el lector puede leer el artículo de David A. Wheeler en http://www.dwheeler.com/oss_fs_why.html.

Capítulo 3

GATE

3.1. Visión general de GATE (General Architecture for Text Engineering)

GATE (General Architecture for Text Engineering) es una infraestructura open-source basada en Java que sirve para desarrollar y reutilizar componentes de software para resolver diversos problemas en el dominio del Procesamiento de Lenguaje Natural. GATE también permite construir y anotar o etiquetar corpus y evaluar las aplicaciones generadas, conectándose a una base de datos de textos. El proyecto GATE está documentado en [11], [7], [8]. Actualmente sigue desarrollándose en la Universidad de Sheffield y está patrocinado por la institución británica Engineering and Physical Sciences Resarch Council (EPSRC). El sitio web de GATE http://www.gate.ac.uk/ proporciona varios servicios que pueden ser útiles para quien se pueda interesar.

GATE es una arquitectura, un framework y un ambiente de desarrollo. Como arquitectura, define la organización de un sistema de ingeniería de lenguaje y la asignación de responsabilidades entre diferentes componentes y asegura que las interacciones entre componentes satisfagan los requerimientos del sistema. Como framework, GATE proporciona un diseño reusable y un conjunto de bloques o módulos que se pueden reusar, extender o adaptar para construir sistemas que procesen lenguaje natural. Por último, como ambiente de desarrollo gráfico, ayuda al usuario a disminuir el tiempo de desarrollo. Esta característica corresponde con la herramienta builder del modelo Java Beans. Claramente, GATE usa el modelo de componentes de software desarrollado por Java Soft.

El framework GATE está compuesto de una biblioteca (library) central

y un conjunto de módulos reusables (como el tokenizador, el delimitador de oraciones, y el analizador morfosintáctico o Part-Of-Speech Tagger). Estas herramientas permiten que el desarrollador no tenga que volver a programarlas cada vez que las necesite. Es decir, GATE nos da un punto de partida para construir aplicaciones más complejas.

Es importante aclarar, sin embargo, que algunos componentes de GATE son dependientes del idioma inglés. En particular, los módulos de Extracción de Información, precisamente el problema abordado en esta tesis. Debido a esa restricción, para poder utilizar GATE para Extracción de Información de textos en español, fue necesario programar el módulo de análisis morfosintáctico para el español [1] como un primer paso para poder utilizar la arquitectura de GATE para extracción de información de textos en este idioma (ver figura 1.1).

Los módulos de GATE se clasifican en Recursos de Lenguaje, Recursos de Procesamiento y Recursos Visuales. Los Recursos de Lenguaje son entidades tales como lexicones y corpus. Los de Procesamiento son entidades primordialmente algorítmicas, como parsers y tokenizadores. Los Recursos Visuales son los componentes usados en la interfaz gráfica. De esta forma GATE separa los datos, los algoritmos y las formas de visualizarlos.

3.2. El modelo de componentes de GATE

La arquitectura GATE está basada en componentes (módulos) de software con interfaces bien definidas que permiten su acoplamiento para formar sistemas modulares de software usables en distintos contextos.

Los componentes de GATE pueden implementarse en varios lenguajes de programación, pero deben ser llamados y reconocidos por el sistema como una clase de Java. Esta clase puede ser el componente en sí o bien ser un wrapper que sirva como enlace para acoplar el componente con GATE.

3.2.1. Wrapper

Un wrapper es un objeto que encapsula a otro para para permitir la comunicación entre el objeto encapsulado y un programa llamador.

Un wrapper actúa como interfaz entre su llamador y el código encapsulado o envuelto. Esta interfaz puede ser necesaria principalmente por tres razones:

[1] De hecho, está pensado para usarse en cualquier idioma, pero yo lo entrené para el español.

- Por compatibilidad (el wrapper está orientado a expandir la funcionalidad del programa encapsulado), por ejemplo si el código envuelto está en un lenguaje de programación diferente o usa diferentes convenciones de llamado.
- Por seguridad, por ejemplo para evitar que el programa llamador ejecute ciertas funciones.
- Para emulación (orientada a la amabilidad), por ejemplo un API DirectX esconde las funciones del driver de la tarjeta de video.

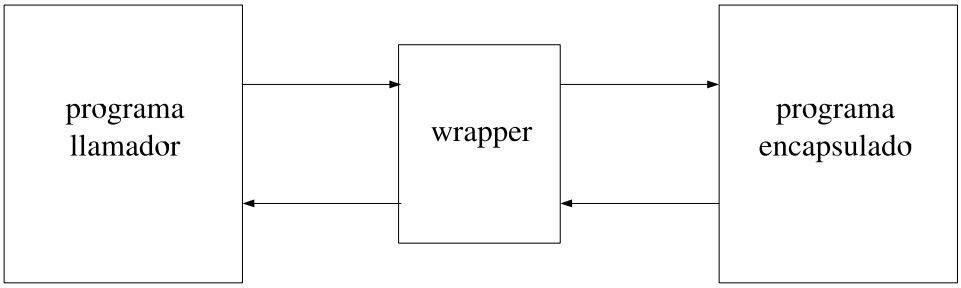

Figura 3.1: wrapper

Esta función de interfaz se logra pasando una referencia (que apunta al objeto a encapsular) al constructor del wrapper, el cual lo guarda en algún lugar. Después, cuando el wrapper recibe una petición del programa llamador (que incluye un cierto input), el wrapper reenvía la petición hacia el objeto encapsulado usando la referencia almacenada. Y una vez que el programa encapsulado ejecuta la tarea el output lo recibe el wrapper reenviándolo al programa llamador. Si la función de interfaz es por compatibilidad entonces el wrapper cumple también la función de traductor: traduce el input a "lenguaje entendible" por el objeto encapsulado y el output de este a "lenguaje entendible" por el programa llamador. En cierto sentido, el wrapper preprocesa el input para que éste sea aceptable para el programa envuelto o encapsulado y postprocesa el output antes de entregarlo al programa llamador.

3.2.2. GATE = GDM + GGI + CREOLE

El modelo de componentes de GATE está basado en el modelo Java Beans (Ver capítulo 2). A continuación se explica brevemente el modelo de GATE mediante el infograma de la figura 3.2.

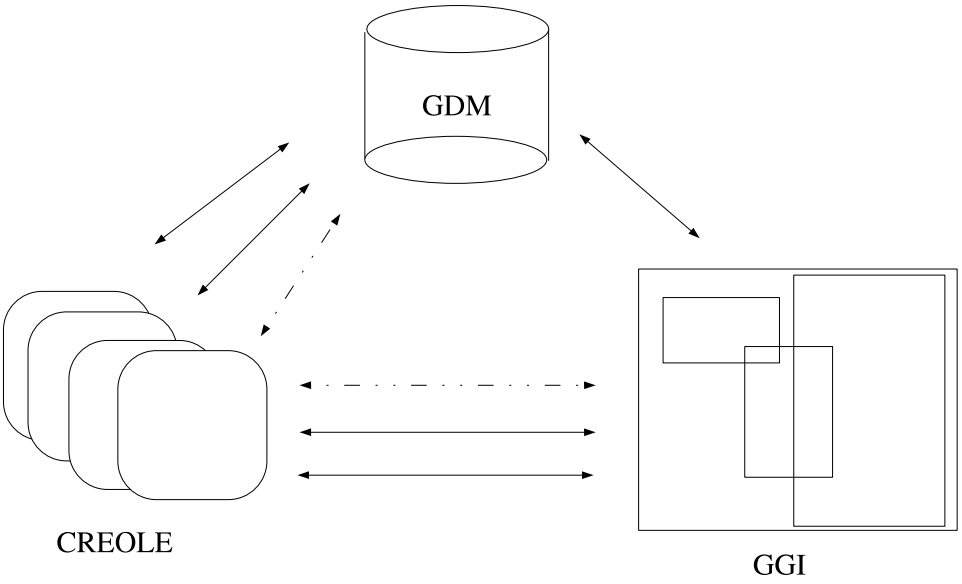

Figura 3.2: Arquitectura de GATE

Como se indica en la figura, GATE consiste de tres partes: el administrador de documentos de GATE (GATE Document Manager, GDM), la interfaz gráfica de GATE (GATE Graphical Interface, GGI) y una colección de objetos (componentes o módulos[2]) reusables para la ingeniería del lenguaje (a Collection of Reusable Objects for Language Engineering, CREOLE). En la terminología bean, CREOLE estaría constituido por beans y el GGI sería la herramienta builder. Es importante aclarar, sin embargo, que la herramienta builder GGI es opcional en GATE pues el usuario puede elegir la interfaz de NET Beans como un builder de más bajo nivel para desarrollar aplicaciones usando los módulos de GATE y correrlas dentro o fuera de GATE.

GDM

El GDM sirve como un centro de comunicaciones para los componentes de GATE. Según Cunningham[6], con GDM como centro de comunicaciones, se reduce el número de interfaces entre n componentes en serie a una sola (GDM), en contraste con n-1 si las componentes tuvieran que comunicarse entre si. El GDM también sirve para almacenar los textos de insumo para

[2]La documentación de GATE habla de objetos para referirse a los elementos de creole, pero realmente son componentes en el sentido de la arquitectura Java Beans.

las aplicaciones de PLN (hechas en GATE) así como la información que es generada al procesarlos. Es también el encargado de establecer (imponer) las restricciones del formato I/O de los componentes CREOLE. Un módulo CREOLE tiene que ser capaz de obtener información del GDM y escribir allí sus resultados una vez que se corrió.

CREOLE

CREOLE está constituido por los módulos o beans a partir de los cuales el usuario puede crear aplicaciones o usar las disponibles para procesamiento de lenguaje natural. Siguiendo el modelo Java Beans, los componentes CREOLE deben tener la propiedad de introspección, es decir, los componentes CREOLE deben poder mostrar sus propiedades ante el GGI. Para esto, la arquitectura GATE utiliza el lenguaje de marcado XML para la configuración de los recursos CREOLE (y para configurar GATE mismo). Es decir, todos los recursos CREOLE tienen metainformación asociada en un documento XML llamado creole.xml. En este archivo se especifica el conjunto de parámetros que un componente CREOLE entiende, cuáles son obligatorios y cuáles son opcionales; también se incluyen los parámetros default de cada componente.

A continuación se muestra el archivo creole.xml del componente CREOLE Tokeniser.

```
<?xml version="1.0"?>
 <!-- creole.xml for the internal tokeniser -->
<CREOLE-DIRECTORY>
 <RESOURCE>
   <NAME>
GATE Unicode Tokeniser
   </NAME>
   <CLASS>
gate.creole.tokeniser.SimpleTokeniser
   </CLASS>
   <COMMENT>
A customisable Unicode tokeniser.
   </COMMENT>
   <PARAMETER NAME="document"
       COMMENT="The document to be tokenised"
  RUNTIME="true">gate.Document</PARAMETER>
   <PARAMETER NAME="annotationSetName"
```

```
        RUNTIME="true"
        COMMENT="The annotation set to be used for
        the generated annotations"
OPTIONAL="true">java.lang.String
    </PARAMETER>
    <PARAMETER
DEFAULT="gate:/creole/tokeniser/DefaultTokeniser.rules"
COMMENT="The URL for the rules file" SUFFIXES="rules"
NAME="rulesURL">java.net.URL
    </PARAMETER>
    <PARAMETER
        DEFAULT="UTF-8"
        COMMENT="The encoding used for reading the definitions"
        NAME="encoding">java.lang.String
    </PARAMETER>
    <ICON>shefTokeniser.gif</ICON>
  </RESOURCE>
</CREOLE>
</CREOLE-DIRECTORY>
```

Para crear un nuevo recurso CREOLE hay que seguir los siguientes pasos:

- Escribir una clase Java que implemente el modelo de componentes de GATE.
- Compilar la clase, junto con todas las clases utilizadas, en un archivo JAR (Java ARchive)
- Escribir los datos de configuración en un archivo xml para este nuevo componente.
- Decirle a GATE la dirección URL de los nuevos archivos JAR y XML que se acaban de crear.

GGI

El GGI es una herramienta gráfica que posibilita al usuario construir y correr componentes y sistemas de PLN (ver figura). El GGI permite crear, visualizar, y editar las colecciones de documentos que están administradas por el GDM y que forman el corpus que los sistemas de PLN en GATE usan como insumo. El GGI también permite visualizar los resultados generados

al correr componentes CREOLE, es decir, permite ver los cambios de las etiquetas asociadas a las palabras del corpus. Además, dentro de la interfaz gráfica de GATE, se pueden realizar distintas tareas como integrar nuevos componentes, inicializar componentes, cambiar los parámetros de componentes (por ejemplo, decirle a un etiquetador morfosintáctico que utilice un lexicón distinto al predeterminado para etiquetar), combinar componentes mediante un pipeline para formar componentes más complejos, evaluar resultados, etc.

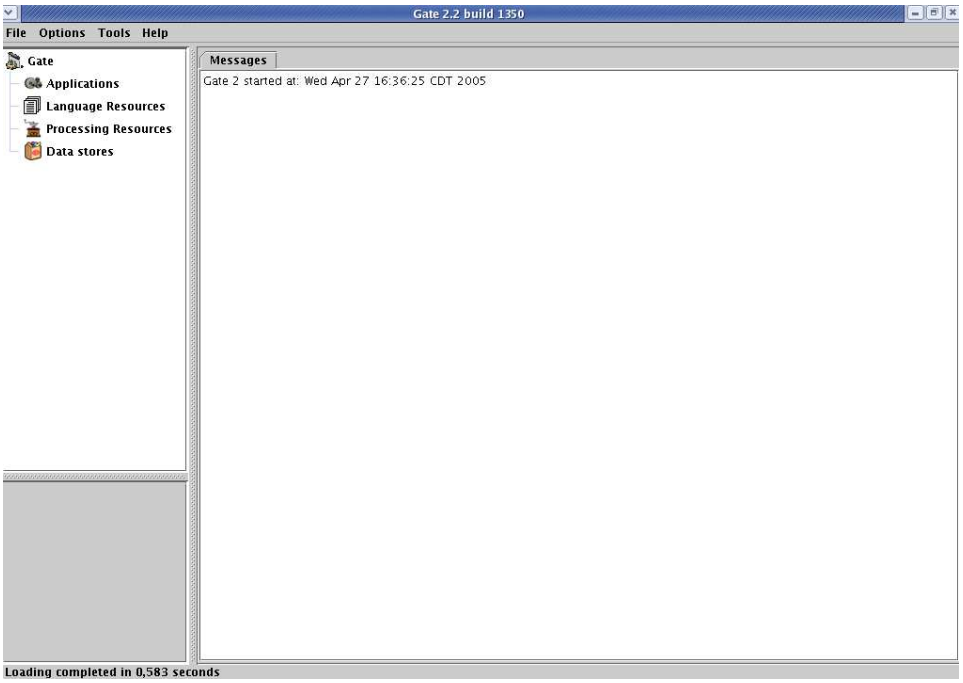

Figura 3.3: GGI de GATE

3.3. Aplicaciones y Bases de Datos

Aparte de los recursos de lenguaje y de procesamiento, GATE tiene otras dos opciones en la paleta: Applications y Data Stores. Las aplicaciones en GATE son conjuntos de recursos de procesamiento agrupados y ejecutados en sucesión sobre algún recurso de lenguaje. La versión actual de GATE (3.0) tiene las opciones: pipeline y corpus pipeline, y para cada una de estas hay una condicionada que va a depender del valor de un atributo en el

documento.

Imaginemos que estamos ante la interfaz gráfica de GATE (ver figura 3.3). Una vez que todos los recursos han sido cargados, una aplicación puede ser creada y corrida. Para ello basta ejecutar un click derecho sobre "Applications" y seleccionar "New", para después elegir entre "corpus pipeline" o "pipeline" o una de las condicionadas. Una aplicación pipeline solamente puede correrse sobre un solo documento, mientras que una del tipo corpus pipeline puede ser corrida sobre todo un corpus.

Para configurar el pipeline[3], se hace doble click sobre la nueva aplicación que se creo usando Applications -> New -> pipeline. De inmediato aparecerán sobre el panel principal dos columnas. En la columna izquierda se encuentran enlistados los recursos de procesamiento que hayan sido cargados (ver figura 3.4). El usuario deberá pasar a la columna derecha los componentes que necesite para poder armar su aplicación (ver figura 3.4). Después, los módulos deben de ponerse en el orden (de arriba hacia abajo) en el que se quiere que se corran. Y, por último, a cada componente del pipeline hay que asignarle el recurso de lenguaje (texto previamente cargado en GATE) que se quiere analizar y hacer click en "Run".

3.4. Anotaciones

Cuando se corren recursos de procesamiento (como tokenizadores, parseadores, etc.) que operan sobre textos, aquellos producen información acerca de éstos. Por ejemplo, cuando se corre un tokenizador, a cada palabra se le asigna un tipo (token-type): word, number, punctuation, etc; cuando se corre un etiquetador morfosintáctico, a cada palabra se le asigna una categoría gramatical (proper noun, verb, etc). Esta información que se produce a partir del texto se representa dentro de GATE como un conjunto de anotaciones.

Una anotación en GATE consiste de:

- un ID, es decir, una identificación única en el documento al que la anotación se refiere
- un type, el cual denota el tipo de anotación. Los diferentes recursos de procesamiento usualmente generan anotaciones de distintos tipos.

[3]Pipeline es un concepto informático que se refiere a la ejecución en sucesión de varios componentes de software donde el output de uno es el input del siguiente. Es un acoplamiento encadenado de las componentes que forman la aplicación.

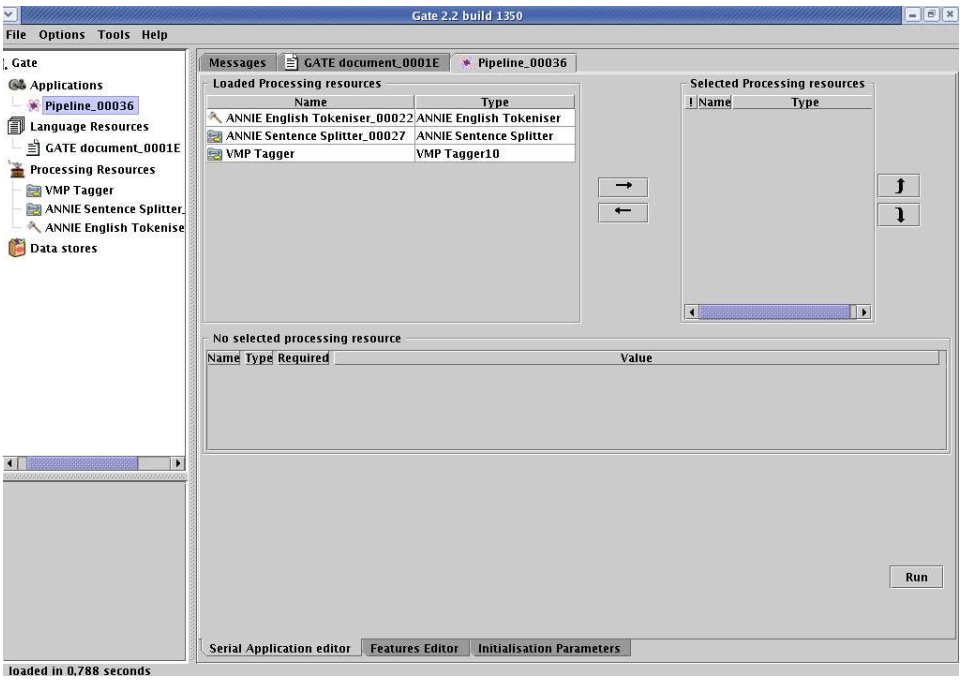

Figura 3.4: Pipeline de GATE. En el lado izquierdo del panel central aparecen todos los recursos de procesamiento que se han cargado.

- StartNode y EndNode. Estos nodos denotan la posición inicial y la posición final que la palabra que se está anotando ocupa en el texto original.

- un conjunto de features en la forma de pares atributo/valor que proporcionan información adicional acerca de la anotación.

Dentro de la interfaz gráfica de GATE, las anotaciones de un texto procesado se pueden ver haciendo doble click en el nombre del texto que se procesó y que aparece en la paleta en Language Resources (ver3.7). Como resultado, en el panel principal aparecerá el texto y en la parte de arriba del panel aparecerán tres botones: text, annotations y annotation sets. Cuando se hace click sobre annotations, en la parte de abajo del panel aparece la tabla de anotaciones. Cuando se hace click en annotation sets, al lado derecho del panel aparecen los nombres de las anotaciones realizadas sobre el texto.

Estas anotaciones junto con el texto se guardan automáticamente en GATE usando la opción Session Persistence que se localiza en Options de

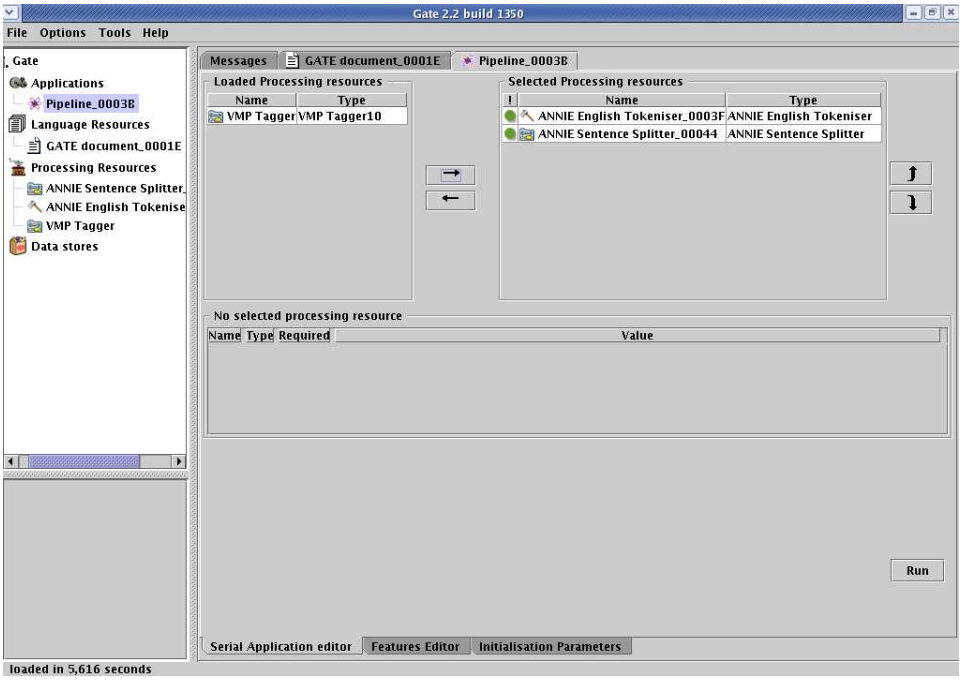

Figura 3.5: GGI de GATE. Los recursos de procesamiento que se quieran utilizar para hacer una aplicación se deberán de pasar a la columna derecha.

la barra de menú. También es posible guardar el texto anotado en formato XML. A continuación se muestra un extracto de un texto en español que fue anotado por tres recursos de procesamiento: el tokenizador ANNIE tokenizer, el delimitador de oraciones ANNIE sentence splitter y el analizador morfosintáctico VMP Tagger.

Observaciones: La anotación es del tipo Token. Los elementos *feature kind*, *lenght*, *orth* fueron producidos por el tokenizador, y el elemento category por el VMP Tagger.

```
<Annotation Type="Token" StartNode="256" EndNode="264">
<Feature>
  <Name className="java.lang.String">string</Name>
  <Value className="java.lang.String">sabíamos</Value>
</Feature>
<Feature>
  <Name className="java.lang.String">kind</Name>
```

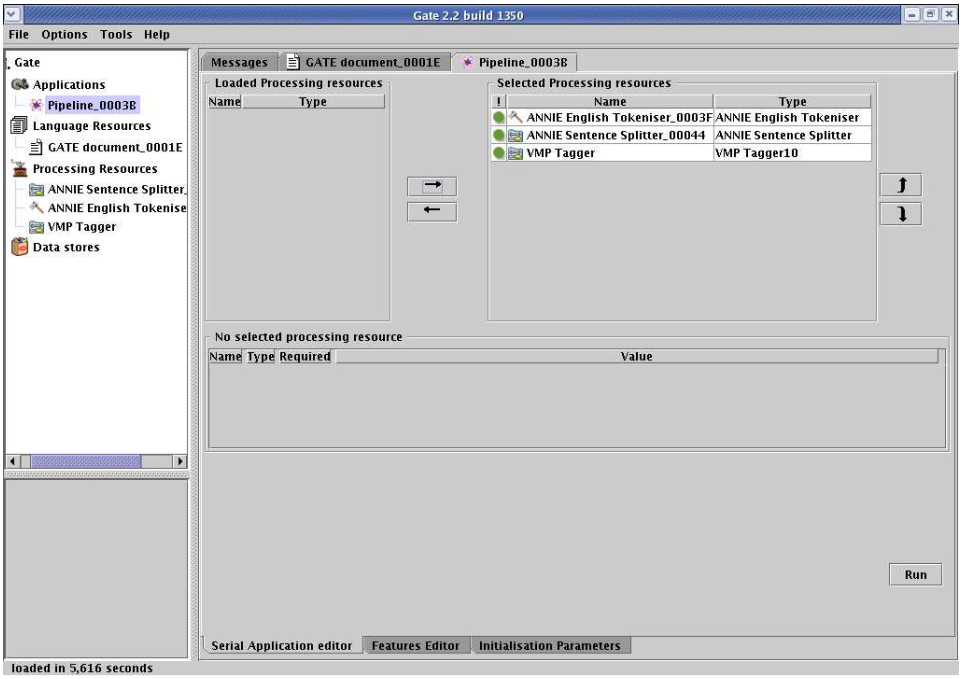

Figura 3.6: GGI de GATE. Los recursos de procesamiento ya están en la columna derecha, listos para ser procesados

```
  <Value className="java.lang.String">word</Value>
</Feature>
<Feature>
  <Name className="java.lang.String">length</Name>
  <Value className="java.lang.String">8</Value>
 </Feature>
<Feature>
  <Name className="java.lang.String">category</Name>
 <Value className="java.lang.String">VMIC1P0</Value>
</Feature>
<Feature>
  <Name className="java.lang.String">orth</Name>
  <Value className="java.lang.String">lowercase</Value>
</Feature>
</Annotation>
```

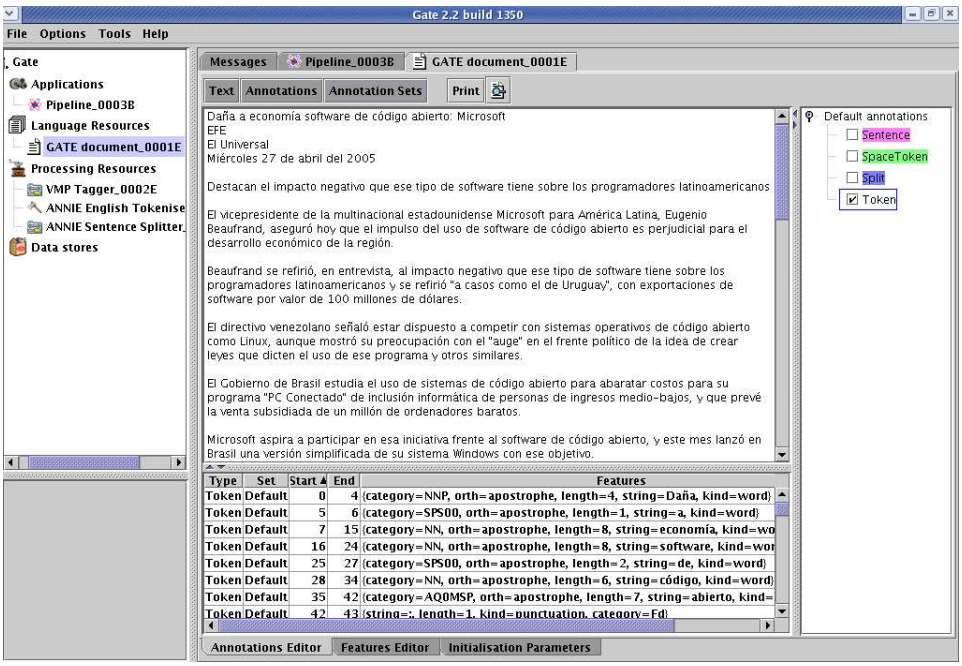

Figura 3.7: GATE GGI Documento procesado con tokenizer, sentence splitter y VMP Tagger. Abajo del texto se puede ver la tabla de anotaciones y a la derecha se puede seleccionar el tipo de anotaciones que se quieren ver.

3.5. JAPE: Expresiones Regulares

JAPE (Java Annotation Patterns Engine) es un módulo de GATE que permite efectuar transformaciones en las anotaciones de un texto previamente anotado (al menos con la anotación token). JAPE utiliza como insumo una gramática JAPE (que está basada en el uso de expresiones regulares) y un texto anotado. La gramática es un conjunto de fases, cada una con reglas patrón/acción, que se ejecutan en forma sucesiva sobre el texto anotado. Las reglas patrón/acción tienen especificado en el lado izquierdo (LI) un patrón de anotación y en el lado derecho (LD) especifican una acción a ser ejecutada si es que el LI se satisface.

El LI de la regla es una expresión regular[4] y como tal puede contener

[4]Expresión Regular es un concepto de teoría de autómatas en donde se demuestra la equivalencia entre expresiones regulares y autómatas finitos (ver Hopcroft[16]). Como se sabe un autómata finito consiste de un alfabeto, un conjunto finito de estados y una función de transición.

los operadores (“*”, “?”, “—”, “+”). El lado derecho de la regla contiene el nombre de la anotación que se quiere agregar al conjunto de anotaciones del texto en caso de que se satisfaga el LI. Junto con esta anotación se pueden especificar los atributos que se quieran asignar a la anotación. (Agreguemos que también es posible que el LD contenga un bloque válido de código Java y es esta propiedad que hace a JAPE una herramienta muy potente y flexible.)

En seguida se presenta un ejemplo del uso de JAPE.

```
Phase: Name
Input: Token
Options: control = appelt

Rule: Orguni
Priority: 25
// University of Sheffield
// University of New Mexico
(
 {Token.string == "University"}
 {Token.string == "of"}
 ({Token.category == "NNP"})+
) :orgName
-->
:orgName.Organization = {rule = "OrgUni"}
```

El ejemplo consta de una sola fase (phase) llamada Name; después se especifica qué tipo de anotaciones recibirá como insumo válido en esta fase. En este caso, solamente las anotaciones Token serán comparadas contra las reglas buscando una correspondencia y todas las otras anotaciones ya generadas en el sistema serán ignoradas. Adicionalmente, las opciones (Options) de la fase le está indicando a JAPE que corra en modo appelt.

Esto significa que si varias reglas son satisfechas por el input en algún momento (es decir, sus lados izquierdos entran en correspondencia con el input), entonces solamente la regla de correspondencia (matching rule) más larga será aplicada; es decir, será aplicada la que entre en correspondencia con la sucesión más larga de anotaciones en el input.

Si hay varias reglas que entran en correspondencia con una región del insumo o input de la misma longitud, entonces es necesario considerar una prioridad de reglas y la regla con la más alta prioridad será aplicada. Otro estilo de procesado es “brill” y cuando JAPE corre en este modo todas las reglas que se satisfacen se aplican. Un tercer estilo es “first”, el cual instruye

a JAPE para que aplique la primera regla que entra en correspondencia con la región de input.

El lado izquierdo de la regla es el bloque entre paréntesis que está antes del símbolo "–>" y contiene una expresión regular que será comparada contra las anotaciones input (en este caso sólo las token) buscando correspondencia. La regla dice que la secuencia de tokens "University", "Of" y uno o más tokens de nombre propio (NNP) debería ser reconocida como un patrón en las anotaciones del texto. La sucesión de anotaciones insumo que satisficieron la regla (entraron en correspondencia con la regla) se llamará de allí en adelante "orgName". Finalmente las proposiciones en el LD (el bloque que está después de la flecha) será ejecutado siempre que el LI es cumplido por el input.

El LD creará una nueva anotación del tipo (type) "Organization" para el fragmento que satisfizo la regla "orgName". La anotación "Organization" tendrá un sólo atributo "rule" el cual será puesto en el valor "OrgUni". Si aplicamos la regla a la secuencia insumo "University", "of" "Sheffield", una nueva anotación del tipo "Organization" será creada para el fragmento "University of Sheffield".

Para aprender más sobre Jape se recomienda leer el capítulo 6 de la Guía de Usuario de GATE.

3.6. ANNIE: a Nearly-New Information Extraction System

GATE se distribuye con un sistema para extracción de información llamado ANNIE que es un conjunto de módulos CREOLE para hacer extracción de información. Sin embargo, ANNIE es dependiente del idioma inglés.

Como se vio en el capítulo 1, para poder hacer un sistema de extracción de información, es necesario contar con los siguientes módulos:

- tokenización
- segmentación de oraciones
- de análisis morfosintáctico
- categorización de sustantivos
- análisis sintáctico de superficie
- reconocimiento de entidades y eventos

- resolución de correferencia
- reconstrucción y generación de plantillas

A continuación se presenta un resumen del análisis que se hizo para determinar cómo se puede adaptar GATE al español para poder hacer sistemas de extracción de información. Se explica brevemente cada módulo ANNIE y se comenta lo que, según investigué, hay que hacer para adaptar ANNIE al español.

3.6.1. Tokenización

ANNIE sí tiene tokenizador. Éste consta de dos partes, una que separa al texto en tokens muy simples como números, símbolos de puntuación, y palabras. Y otra con la que, mediante el uso de JAPE, se pueden construir tokens más complejos. Veamos un ejemplo. El tokenizador por si sólo reconocería a $35.89 como cuatro tokens:

- $ token del tipo Symbol
- 35 token del tipo Number
- . token del tipo Punctuation
- 89 token del tipo Number

Después, usando JAPE, se puede lograr que se reconozca a $35.89 como un sólo token (por ejemplo, Money).

El tokenizador de ANNIE sí se puede reusar para el español sin ningún problema (sin ningún cambio).

3.6.2. Segmentación de Oraciones

ANNIE también cuenta con un módulo para delimitar las oraciones de un texto: ANNIE Sentence Splitter. Éste módulo también se puede reusar para el español sin ninguna modificación.

3.6.3. Análisis Morfosintáctico

ANNIE cuenta también con un módulo de análisis morfosintáctico: Hepple Part-Of-Speech Tagger. Pero este módulo es dependiente del idioma inglés. De hecho, es dependiente del conjunto de etiquetas Penn Treebank (Ver la sección 4.3 en la página 55). Esté módulo de GATE no es open-source,

por lo que no es adaptable para el español (pues su código es cerrado). Este módulo es el que sustituye mi VMP Tagger (ver capítulo 5)

3.6.4. Categorización de Sustantivos

Este análisis se puede hacer usando el módulo de ANNIE llamado Gazetteer. El módulo ANNIE Gazetteer recibe un archivo llamado list.def. Este archivo enlista los nombres de los archivos que contienen listas de palabras clasificadas. Los renglones del archivo list.def son de la forma nombreDelArchivo.lst:categoria:subcategoría.

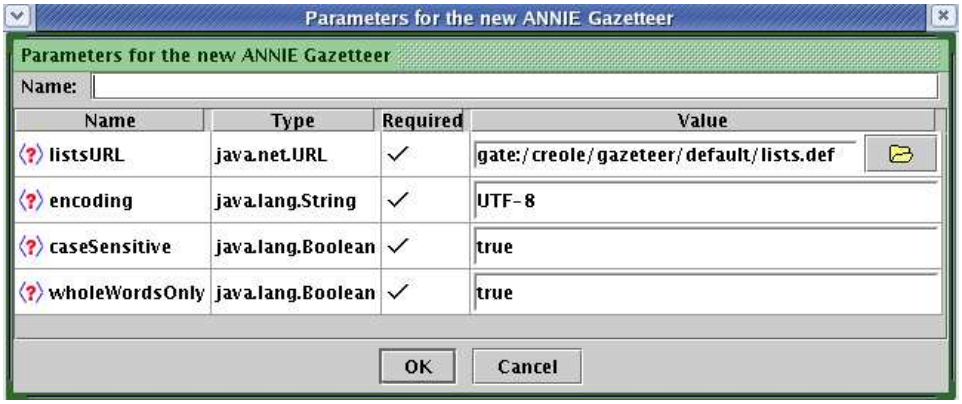

Figura 3.8: ANNIE Gazetteer

Se muestra un extracto de la lista abajo:

```
abbreviations.lst:stop
cdg.lst:cdg
charities.lst:organization
city.lst:location:city
city_cap.lst:location:city
company.lst:organization:company
company_cap.lst:organization:company
country.lst:location:country
country_cap.lst:location:country
```

Ahora se muestra un extracto de el archivo city.lst

```
Aaccra
Aalborg
```

```
Aarhus
Ababa
Abadan
Abakan
Aberdeen
Abha
Abi Dhabi
Abidjan
Abilene
Abu
Abu Dhabi
Abuja
Acapulco
```

Por ejemplo, la palabra *Acapulco* está en la lista city.lst que a su vez está en la lista list.def como city.lst:location:city (es decir, city.lst contiene palabras pertenecientes a la categoría *location* y a la subcategoría *city*).

Si se escribe en un archivo de texto el enunciado *I lived in Acapulco many years ago.*, la información de que *Acapulco* es una palabra de la categoría *location* y subcategoría *city* quedará guardada dentro GATE una vez que se corra el módulo ANNIE Gazetteer sobre el enunciado. En la figura 3.9 se puede apreciar que la palabra Acapulco (que empieza en el lugar 11 y termina en el 19) pertenece a la categoría (*majorType*) *location* y a la subcategoría *minorType city*.

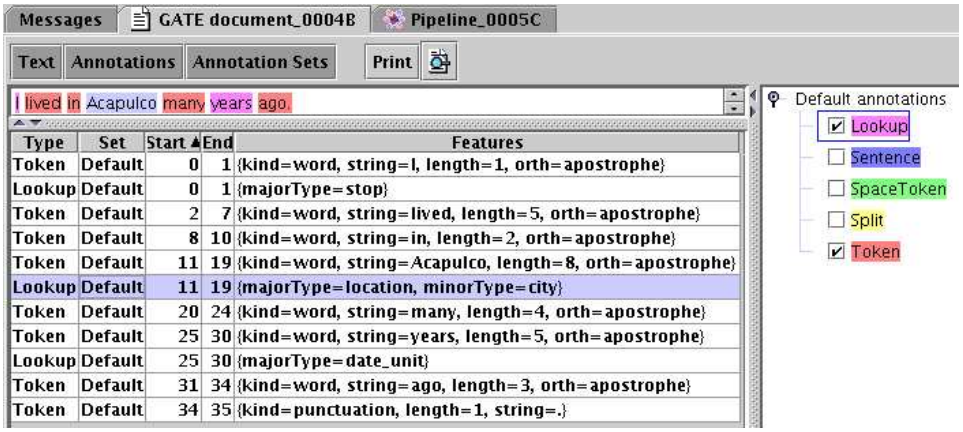

Figura 3.9: ANNIE Gazetteer

Si se logran conseguir muchas listas de palabras para el español, lo cual

puede ser más o menos fácil de encontrar en la Red, se podrá reutilizar el módulo Gazetteer de ANNIE. Las categorías y subcategorías para etiquetar se pueden modificar también. Es decir, el módulo Gazetteer de GATE es totalmente adaptable para el español, porque la forma de categorizar a los sustantivos es buscándolos en las listas de palabras. Entonces, si se logra conseguir o hacer listas de palabras clasificadas para el español, se podrá reusar el módulo Gazetteer.

3.6.5. Análisis Sintáctico de Superficie

Recordemos que en el análisis sintáctico de superficie, se busca reconocer a los sintagmas nominales y verbales de la oración. GATE cuenta con un módulo para reconocer los sintagmas verbales: ANNIE Verb Group Chunker. (Ver figura 3.10.) Este módulo está basado en gramáticas JAPE para reconocer a los sintagmas verbales. Entonces, para poder reusar este módulo para el español, hay que reescribir las reglas JAPE para el español.

En la versión de GATE (versión 2) con la que se trabajó esta tesis, no se incluye ningún módulo para reconocer sintagmas nominales. La versión 3 de GATE (que se liberó el 14 de enero de 2005) sí cuenta con un módulo llamado Noun Phrase Chunker. Por razones de tiempo y dado que ya tenía mi módulo VMP Tagger para la versión 2, la tres no la he analizado con detalle.

3.6.6. Reconocimiento de Entidades y Eventos

GATE cuenta con un módulo para reconocimiento de entidades: ANNIE Named Entity Transducer. Este módulo recibe como parámetro un archivo main.jape, el cual contiene una lista de gramáticas JAPE. Estas serán procesadas (en el orden en que se escriban en la lista) por el módulo ANNIE Named Entity Transducer. (Ver figuras 3.11 y 3.12).

Este módulo también se puede reutilizar siempre y cuando se adapten al idioma español las reglas que reconocen las entidades.

3.6.7. Resolución de Correferencia

La tarea de resolución de correferencia trata de identificar equivalencias entre entidades que fueron reconocidas en la fase de Reconocimiento de Entidades. Esta tarea ayuda a la determinación apropiada de atributos y relaciones entre entidades.

Hay diferentes tipos de correferencia pero no todos son igualmente importantes en términos de su ocurrencia. En lingüística la correferencia más

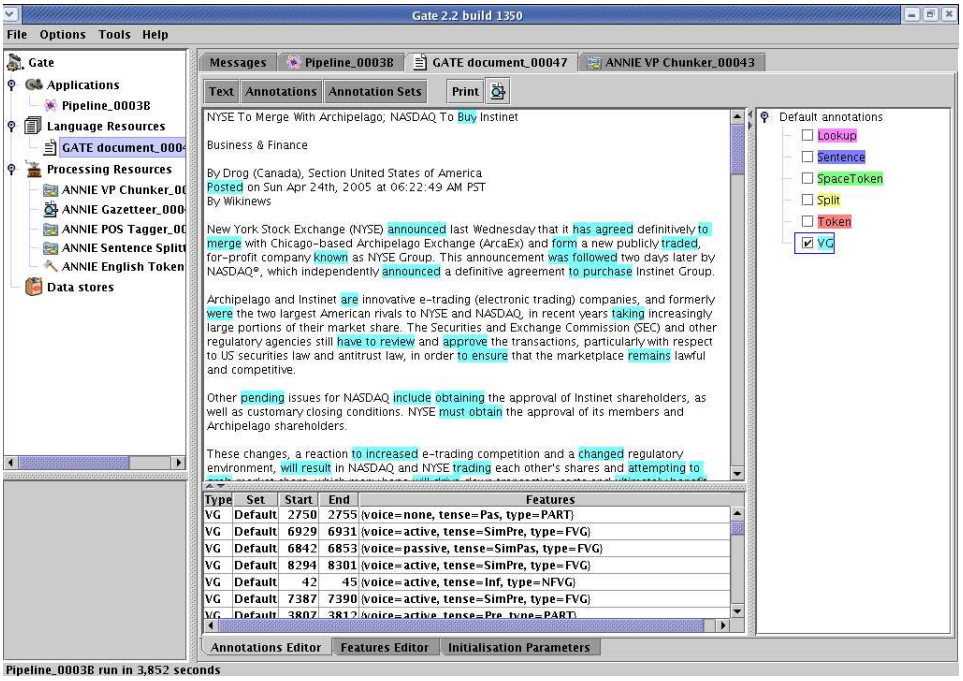

Figura 3.10: ANNIE Verb Group Chunker

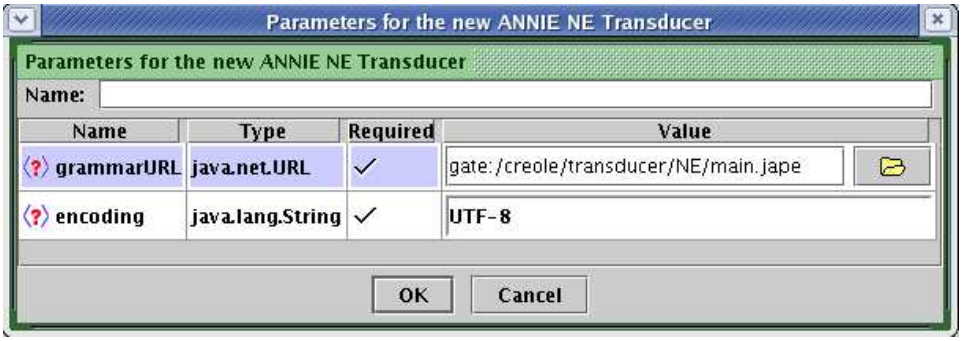

Figura 3.11: ANNIE Named Entity Transducer

conocida es la anáfora y denota el fenómeno de referencia a una entidad ya mencionada en el texto, de una forma diferente a como se mencionó primero. Lo más común es utilizar un pronombre o un nombre diferente. De esta manera la anáfora es una referencia a una entidad ya mencionada o antecedente.

En procesamiento de lenguaje natural se denomina anáfora a la ocurren-

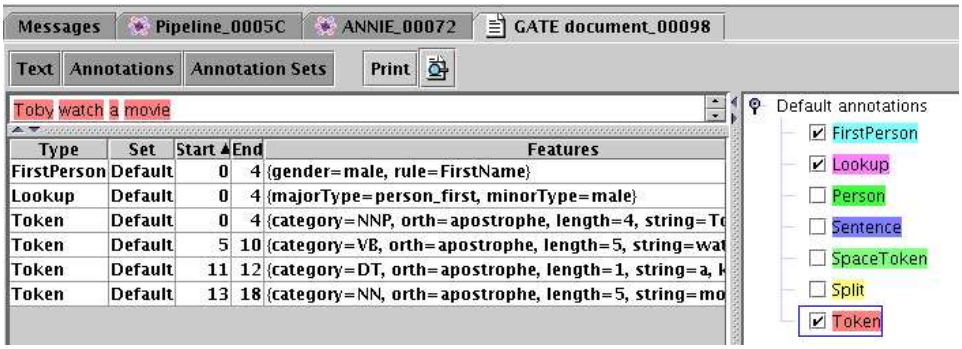

Figura 3.12: ANNIE Named Entity Transducer

cia de la entidad con otros nombres. El proceso de encontrar el nombre de la entidad antecedente para cada anáfora en un texto se denomina resolución de anáfora. Una expresión de lenguaje natural usada para hacer una referencia es denominada una expresión que refiere (referring expression) y la entidad a la que se refiere es denominada referente. Dos expresiones que refieren y usadas para referirse a la misma entidad se dice que correfieren.

GATE cuenta con un módulo de resolución de anáfora que tiene dos opciones: nominal y pronominal. El submódulo de correferencia pronominal ejecuta la resolución de anáfora utilizando el formalismo de la gramática JAPE. El submódulo de correferencia nominal no está documentado en la Guía de Usuarios de GATE. Otros tipos de correferencia posiblemente se agreguen en el futuro dada la estructura modular de GATE.

El módulo de correferencia depende de las anotaciones creadas por los otros módulos de ANNIE:

- Token
- Sentence Splitter
- Hepple POS Tagger
- NE transducer
- OthoMatcher

Como el módulo de correferencia utiliza gramáticas JAPE, entonces este módulo se puede reutilizar adaptando las gramáticas para español.

3.6.8. Reconstrucción y Generación de Plantillas

GATE no cuenta con ningún módulo para generar plantillas.

Casi todos los módulos utilizan JAPE y son en consecuencia reusables adaptándolos para el español pero esta adaptación no es una tarea fácil.

3.7. Instalación de GATE

Como ya se dijo GATE está programado en Java. Sin embargo no es necesario tener instalado el lenguaje Java para poder instalar y correr GATE pues la instalación se puede hacer con un instalador específico de plataforma (denominado platform-specific installer) asociado y trae incorporado el JVM (Java Virtual Machine) que corre los ejecutables de GATE. En la página de GATE, al entrar, detectan tu plataforma y te recomiendan un instalador. Pero también hay una forma difícil de instalar GATE compilándolo, lo cual si requiere tener instalado el lenguaje Java. Y si se quiere explotar todo el potencial de GATE es indispensable tener instalado Java para desarrollar módulos acoplables a esa arquitectura.

3.7.1. Bajando Java y GATE

Java se puede bajar de:
`http://java.sun.com/j2se/1.4.2/download.html`

Para instalar java es necesario bajar el archivo
`j2sdk-1_4_2_05-windows-i586-p.exe` (51.09 MB para Windows)
o `j2sdk-1_4_2_05-linux-i586-rpm.bin` para Linux.

Para bajar GATE hay que visitar la página http://gate.ac.uk/, ir a la sección de download, llenar una forma que te piden y recibirás una dirección de ftp para que puedas bajar el sistema.

3.7.2. Instalando Java 2 SDK, Standard Edition 1.4.2_05

En Windows hay que correr el archivo exe que ya se debe tener en disco obteniéndolo como se indica arriba. Para instalar Java en Linux es recomendable seguir paso a paso alguno de los tutoriales existentes en Internet. Yo usé uno que se localiza en `http://saladelfrio.com/node/view/12` El proceso de instalación que yo seguí lo resumo en seguida:

1. `\#sh j2sdk-1_4_2_05-linux-i586-rpm.bin`
2. `\#sh rpm -ivh j2sdk-1_4_2_05-linux-i586-rpm`
3. `Hacer el archivo /etc/profile.d/java.sh con el siguiente`

```
contenido
export JAVA_HOME="/usr/java/j2sdk1.4.2_05"
export JAVA_PATH="$JAVA_HOME"
export PATH="$PATH:$JAVA_HOME/bin"
export JDK_HOME="$JAVA_HOME"
4. Asignar permiso de ejecución sobre el archivo java.sh
chmod +x /etc/profile.d/java.sh
```

3.7.3. Instalando GATE

Como se sabe, Java es un lenguaje multiplataforma. En consecuencia GATE puede ser instalado en Windows, o en Linux (y en Mac y otras). En Windows la instalación de GATE es tan sencilla como hacer doble click en el archivo gate.exe y seguir las instrucciones.

En Linux (que es donde yo lo instalé) se debe correr en la consola (intérprete de comandos o shell) lo siguiente:

```
\#sh gate.bin
```

Con este comando se corre el instalador y se siguen las instrucciones.

Puede ser de utilidad para los interesados en instalar GATE la siguiente experiencia que adquirí durante la instalación. Cuando yo instalé GATE y traté de ejecutar `sh gate.bin` me aparecieron los siguientes mensajes:

```
Preparing to install...
Extracting the installation resources from
the installer archive...
Configuring the installer for this system's
environment...
No Java virtual machine could be found from
your PATH environment variable. You must
install a VM prior to running this program.
```

El error me sorprendió porque se suponía que ya había instalado JVM. Por eso busqué en Google palabras clave de este mensaje esperando que alguien hubiera entrado a un grupo de discusión con un problema parecido. Y ¡sí! Alguien tenía el mismo error que yo (ver foro de Linux Questions http://www.linuxquestions.org/questions/history/204361.) Descubrí que había que agregar en el PATH a java. Para eso, tuve que modificar mi `bash_profile`. Para los que no sepan qué es eso, sugiero visitar la página Changing your Bash Profile. Esta solución no está documentada en la guía de usuario de

GATE[11]. La solución sugerida en el foro sí funciona. Es la siguiente. Usando gedit se agregan las líneas siguientes al archivo bash_profile (suponiendo que tengas instalado JVM en /usr/local/java) :

```
PATH=/usr/local/java/bin:\$PATH
export PATH

JAVA\_HOME=/usr/local/java
export JAVA\_HOME
```

Una vez instalado GATE se corre el archivo gate.bat si se está en Windows o `# ./gate.sh`

Capítulo 4

Etiquetadores Morfosintácticos

4.1. Partes del Discurso (Categorías Gramaticales)

Las palabras de un lenguaje natural tradicionalmente se han agrupado en clases de equivalencia llamadas partes del discurso, partes de la oración o categorías gramaticales. Esta clasificación se basa en las funciones morfológica y sintáctica de la palabra, es decir, las clases de palabras se reconocen tanto por su función en la oración como por sus características morfológicas (de la estructura interna de la palabra).

La morfología se ocupa de la estructura interna de la palabra. La sintaxis se encarga de regular la combinación de palabras dentro de la oración. (Para la morfología la palabra es la unidad máxima y para la sintaxis la mínima.)

En español usualmente se tienen las siguientes categorías gramaticales:

1. sustantivos: *gato, pluma, libro, etc*
2. verbos: *correr, ganó, presentaste, etc*
3. adjetivos: *bonito, azul, limpio, etc*
4. adverbios: *sí, no, ahora, rápidamente, etc*
5. artículos: *el-los , la-las, lo / un-unos, una-unas, etc*
6. pronombres: *yo, nosotros, tú, ustedes, éste, ése, aquél, etc*

7. preposiciones: *en, con, por, desde, etc*
8. conjunciones: *y, o, pero, etc*
9. interjecciones: *¡oh!, ¡ay!, etc*

Las partes del discurso se dividen en dos categorías: las clases abiertas y las clases cerradas.

- Clases cerradas. En esta categoría se encuentran las partes del discurso que tienen un número casi fijo de palabras (preposiciones, artículos, pronombres, conjunciones, e interjecciones).
- Clases abiertas. En esta categoría se encuentran los verbos, sustantivos, adverbios y adjetivos.

Algunas palabras pueden variar su forma dependiendo de las construcciones sintácticas de que forman parte [1]. De acuerdo a este criterio, las categorías gramaticales también se puede clasificar en dos grupos: las variables y las invariables. Las categorías gramaticales o partes de la oración variables son aquéllas que tienen accidentes gramaticales (o morfemas flexivos). A continuación se presentan los cuadros de los dos tipos de accidentes gramaticales.

- NOMINALES: afectan al artículo, al sustantivo, al adjetivo y al pronombre.
  - Género: masculino y femenino
  - Número: singular y plural
- VERBALES: afectan al verbo.
  - Voces: activa y pasiva
- Modos:
  - personales: indicativo, subjuntivo, imperativo
  - impersonales (formas nominales o verboides): participio, infinitivo y gerundio

[1] En lingüística, la flexión es la alteración que experimentan las palabras, usualmente mediante afijos o desinencias, para expresar sus distintas funciones dentro de la oración y sus relaciones de dependencia o de concordancia con otras palabras o elementos oracionales. La conjugación y la declinación son formas de flexión.

- Tiempos (nomenclaturas de la Real Academia y de Andrés Bello):
  - simples:
    - presente
    - pretérito imperfecto o copretérito
    - pretérito indefinido o pretérito
    - futuro imperfecto o futuro
  - compuestos:
    - pretérito perfecto o antepresente
    - pretérito anterior o antepretérito
    - pretérito pluscuamperfecto o antecopretérito
    - futuro perfecto o antefuturo
- Número y personas:
  - singular: 1a (yo), 2a (tú, usted), 3a (él, ella)
  - plural: 1a (nosotros), 2a (vosotros, ustedes), 3a (ellos, ellas)

A continuación se enlistan las partes del discurso del español.

- sustantivo
- verbo
- adjetivo
- adverbio
- preposición
- pronombres
- artículo
- conjunción
- interjección

4.2. Etiquetadores Morfosintácticos (Part-Of-Speech Taggers)

Hemos traducido aquí Part-Of-Speech Tagger como etiquetador morfosintáctico, siguiendo a Alonso[1], pues estos etiquetadores están en la frontera entre la estructura interna de la palabra y su función (entre la morfología y la sintaxis). Por eso nos parece poco adecuado llamarlos etiquetadores morfológicos. Como se vio en el capítulo 1 (ver figura 1.1), el pos tagger está, en el modelo presentado allí, después del tokenizador y el sentence splitter pero antes de categorización de sustantivos al cual le sigue el análisis sintáctico de superficie (shallow parsing).

Un etiquetador morfosintáctico se encarga de etiquetar cada palabra de un texto con su categoría gramatical (lexical class) y otras marcas léxicas (ver cuadros 4.3 a 4.13) como modo, tiempo, persona para verbos; género, número para sustantivos; coordinada o subordinada para conjunciones etc. También se acostumbra etiquetar a los signos de puntuación.

Un etiquetador morfosintáctico recibe un texto pretokenizado[2] y un conjunto de etiquetas posibles[3] y devuelve el texto etiquetado con la mejor etiqueta para cada palabra. Usualmente se etiqueta de la siguiente forma: palabra/etiqueta. GATE tiene un método particular de registrar las etiquetas basado en anotaciones.

Enseguida se muestran dos ejemplos tomados del corpus CLIC-TALP que demuestran tanto la forma de etiquetar que es ya un estándar en el español, como las ambigüedades que pueden presentarse. Es el único corpus etiquetado manualmente al que tuve acceso. Fue elaborado por Montserrat Civit[5].

```
joven/NCCS000 progresista/AQ0CS0 algo/RG desaliñada/AQ0FSP

el/DA0MS0 joven/AQ0CS0 equipo/NCMS000 actual/AQ0CS0

después_de/SPS00 haber/VAN0000 estado/VMP00SM años/NCMP000
combatiendo/VMG0000 la/DA0FS0 guerrilla/NCFS000

en/SPS00 un/DI0MS0 estado/NCMS000 de/SPS00 feliz/AQ0CS0
inconsciencia/NCFS000
```

[2]Así lo exige el módulo de Brill (ver sección 4.4). El etiquetador de GATE exige un texto que ya pasó por el tokenizer pero también por el sentence splitter, es decir, ya con las anotaciones del tipo token y del tipo sentence.

[3]que, en el caso del Brill tagger, vienen implícitas en las reglas de transformación

Las etiquetas refieren a los cuadros del 4.3 al 4.13. Por ejemplo NCCS000, la etiqueta de "joven", significa nombre común común singular. La primera letra de la etiqueta manda al cuadro correspondiente y las siguientes corresponden a las opciones de columna de ese cuadro. En estos ejemplos se pude ver que la asignación automática de una etiqueta a cada palabra no es fácil. En ellos, las palabras joven y estado son ambiguas. Es decir, cumplen más de una función en el lenguaje natural del español.

La desambiguación por parte de los hablantes o usuarios del idioma se realiza por el contexto. La palabra joven cumple la función de sustantivo (nombre, en la terminología del CLIC-TALP) en la primera oración, y de adjetivo en la segunda. En la tercera oración la palabra estado es un verbo y en la cuarta es un sustantivo.

En la terminología del procesamiento de lenguaje natural, se distingue entre type y token. La palabra joven es el type y su ocurrencia en el texto es el token. Un etiquetador morfosintáctico etiqueta los tokens, es decir, las ocurrencias concretas de los types, y tiene que decidir cual es la función que está cumpliendo ese token dentro de la oración. Nótese que la desambiguación en este nivel es diferente que en la resolución de anáfora; en el POS Tagging se desambigua sobre la función de la palabra en una oración (la desambiguación es sintáctica), en la resolución de anáfora la desambiguación es sobre significado (es desambiguación semántica).

A pesar de que, desde el punto de vista sintáctico, la mayoría de las palabras del español son inambiguas (sólo tienen una etiqueta posible), es muy posible que muchas de las palabras de esa minoría ambigua sean las más usadas en el español. Por lo menos ese es el caso en el idioma inglés. Los resultados de DeRose[9] se muestran en el cuadro 4.1.

Yo obtuve una tabla similar a la de DeRose para el español con un programa en python aplicado al lexicón que fue obtenido del entrenamiento del Brill Tagger con el corpus CLIC-TALP (el número de palabras del lexicón es 16,288). Los resultados son similares a los obtenidos por DeRose para el inglés. (Ver cuadro 4.2.)

Los etiquetadores morfosintácticos pueden ser de dos tipos: basados en reglas (modo experto) o basados en métodos estocásticos (modo empírico). Los etiquetadores basados en reglas generalmente usan un conjunto de reglas elaborado por expertos [4]. Un ejemplo de regla podría ser: *etiquetar a la palabra como adverbio si termina con el sufijo -mente*

Los etiquetadores estocásticos utilizan un corpus de entrenamiento (un corpus manualmente etiquetado) para computar la probabilidad de que una

[4]Ver ENGTWOL tagger en http://www.lingsoft.fi/cgi-bin/engtwol

Cuadro 4.1: Números de tipos de palabra (types) en lexicón derivado del corpus Brown. (88.5 % de palabras no ambiguas.)

No. Tags	No. Words
Unambiguous (1 tag)	35340
Ambiguous (2-7 tags)	4100
2 tags	3760
3 tags	264
4 tags	61
5 tags	12
6 tags	2
7 tags	1 ("still")

Cuadro 4.2: Números de tipos de palabra (types) en lexicón derivado del corpus CLIC-TALP

No. de etiquetas	**No. de palabras**	**Porcentaje**
1	15,625 95,9	95.9
2	605 3,7	3.7
3	52 ,32	.32
4	5 0,03	.03
5	1 0,006 ("medio")	.006
	16,288	

Cuadro 4.3: Etiquetas para los adjetivos

Adjetivo	**tipo**	**apreciativo**	**género**	**número**	**participio**
A	Q (calificativo)	A (sí)	M (masculino)	S (singular)	P (sí)
	O (ordinal)	0 -	F (femenino)	P (plural)	0 -
	0 -		C (común)	N (invariable)	

Cuadro 4.4: Etiquetas para los adverbios

Adverbio	**tipo**
R	G (general)
	N (negativo)

palabra tenga una cierta etiqueta dentro de un cierto contexto [5].

4.3. Conjunto de Etiquetas Usado en el Corpus CLIC-TALP

De acuerdo a Jurafsky [21], los conjuntos de etiquetas más usados para el inglés son el Penn Treebank[6] [24] que consta de 45 etiquetas y es el que usa el corpus Brown[7]; el C5 de 61 etiquetas usado por el proyecto CLAWS para etiquetar el corpus llamado British National Corpus (BNC)[8]; y el C7, de 146 etiquetas, usado para etiquetar un subconjunto del BNC[9]

El conjunto de etiquetas para el español que tuve disponible es el utilizado por el corpus etiquetado CLIC-TALP. Este corpus lo utilicé para entrenar al Brill Tagger (ver capítulo 5). Las etiquetas del corpus CLIC-TALP se describen en los siguientes cuadros del 4.3 al 4.13.

[5] Ver el etiquetador TNT[3] http://www.coli.uni-sb.de/~thorsten/tnt/

[6] Ver http://www.cis.upenn.edu/~treebank/

[7] http://clwww.essex.ac.uk/w3c/corpus_ling/content/corpora/list/private/brown/brown.html

[8] Ver http://www.natcorp.ox.ac.uk/index.html, el BNC es un corpus etiquetado que consta de 100 millones de palabras

[9] El C7 fue usado para etiquetar un subconjunto de 2 millones de palabras del BNC.

Cuadro 4.5: Etiquetas para los determinantes

Determinante	**tipo**	**persona**	**género**	**número**	**poseedor**
D	D (demostrativo)	1 (primera)	M (masculino)	S (singular)	S (singular)
	P (posesivo)	2 (segunda)	F (femenino)	P (plural)	P (plural)
	T (interrogativo)	3 (tercera)	C (común)	N (invariable)	
	E (exclamativo)		N (invariable)		
	I (indefinido)				
	A (artículo)				
	N (numeral)				

Cuadro 4.6: Etiquetas para los nombres

Nombre	**tipo**	**género**	**número**			**apreciativo**
N	C (común)	M (masculino)	S (singular)	0 -	0 -	A (sí)
	P (propio)	F (femenino)	P (plural)	0 -	0 -	
		C (común)	N (invariable)	0 -	0 -	

Cuadro 4.7: Etiquetas para los verbos

Verbo	**tipo**	**modo**	**tiempo**	**persona**	**número**	**género**
V	M (principal)	I (indicativo)	P (presente)	1(primera)	S (singular)	M (masculino)
	A (auxiliar)	S (subjuntivo)	I (imperfecto)	2 (segunda)	P (plural)	F (femenino)
	S (semiauxiliar)	M (imperativo)	F (futuro)	3 (tercera)		
		N (infinitivo)	C (condicional)			
		G (gerundio)	S (pasado)			
		P (participio)				

Cuadro 4.8: Etiquetas para las interjecciones y abreviaturas

Interjección
I
Abreviatura
Y

Cuadro 4.9: Etiquetas para las preposiciones

Preposición	**tipo**	**forma**	**género**	**número**
S (adposición)	P (preposición)	S (simple)		
		C (contraída)	M (masculino)	S (singular)

Cuadro 4.10: Etiquetas para los pronombres

Pron.	**tipo**	**persona**	**género**	**número**	**caso**	**poseedor**	**politeness**
P	P (personal)	1 (primera)	M (masculino)	S (singular)	N (nominativo)	S (sing.)	P (polite)
	D (demostrativo)	2 (segunda)	F (femenino)	P (plur.)	A (acusativo)	P (plural)	
	X (posesivo)	3 (tercera)	C (común)	N (invariable)	D (dativo)		
	T (interrogativo)		N (neutro)		O (oblicuo)		
	R (relativo)						
	N (numeral)						
	E (exclamativo)						

Cuadro 4.11: Etiquetas para las conjunciones

Conjunción	tipo
C	C(coordinada)
	S(subordinada)

Cuadro 4.12: Otras etiquetas

Cifras	Z
Horas y fechas	W
Cantidad monetaria	Zm

Cuadro 4.13: Etiquetas para los símbolos de puntuación

,	Fc	.	Fp	“	Fe
...	Fs	:	Fd	”	Fe
;	Fx	%	Ft	’	Fe
-	Fg	/	Fh	–	Fg
‘	Fe	(	Fpa	)	Fpt
¿	Fia	?	Fit	¡	Faa
!	Fat	[	Fca	]	Fct
{	Fla	}	Flt	≪	Fra
≫	Frc	**Otros**	Fz		

4.4. Etiquetador de Brill y su Módulo de Entrenamiento

Como ya se dijo en el capítulo 3, GATE se distribuye libremente junto con un conjunto de módulos (ANNIE, A Nearly New Information Extraction System) para desarrollar sistemas de Extracción de Información para el inglés. Uno de los módulos de ANNIE es el POS Tagger de Hepple[15]. Este POS Tagger es una versión modificada del etiquetador de Brill y es por eso que elegí estudiar para esta tesis el etiquetador de Brill. Pero además porque Hepple utiliza un subconjunto del input del Brill como entrada[10]. Este es el punto de acoplamiento donde entra el VMP Tagger. El Hepple lo sustituí por el VMP Tagger.

Eric Brill presentó, con su trabajo de tesis de doctorado[6], un nuevo algoritmo llamado Transformation-Based Error-Driven Learning[11]. Con este algoritmo se puede obtener una lista ordenada de transformaciones (reglas de etiquetado), tomando como base de entrenamiento un corpus pequeño etiquetado manualmente; esta lista puede ser usada para etiquetar un texto nuevo (con la misma estructura del etiquetado que el corpus de entrenamiento).

En 1995, Brill presentó una aplicación de su algoritmo[4]: un POS tagger[12] que incluye un módulo de aprendizaje para entrenamiento en idiomas distintos del inglés. Este etiquetador se encuentra disponible libremente en la red[13] bajo la Open Source Initiative (MIT-license[14]) y corre bajo sistemas operativos de la familia Unix (está programado en C y Perl).

El módulo de entrenamiento del etiquetador de Brill consta de dos submódulos que permiten el aprendizaje (en el sentido de la inteligencia artificial) a partir de un corpus de entrenamiento: el submódulo para generar reglas de etiquetación para palabras desconocidas (submódulo léxico según Brill) y el

[10]Lo primero que se intentó fue entrenar al Brill con el corpus CLIC-TALP para obtener cuatro listas: un lexicón, una lista de reglas para etiquetar palabras desconocidas (no en el lexicón), una lista de reglas de contexto y un bigrama. Con esto se pensó que se podría adaptar el Hepple tagger al español. Pero este plan no funcionó porque el conjunto de etiquetas de Hepple es distinto al del corpus CLIC-TALP. Y también porque el código del Hepple Tagger no es abierto (contradiciendo la información encontrada en varios artículos de GATE donde presumen que todo ANNIE es open-source). Por esta razón no hubo forma de "decirle" al programa que reconociera otro tipo de etiquetas

[11]Ver http://www.cs.jhu.edu/∼brill/dissertation.ps

[12]Se ha traducido simplemente como etiquetador y cuando se quiera ser más específico se le llamará etiquetador morfosintáctico.

[13]http://www.cs.jhu.edu/∼brill/RBT1_14.tar.Z

[14]Ver http://www.opensource.org/licenses/mit-license.php

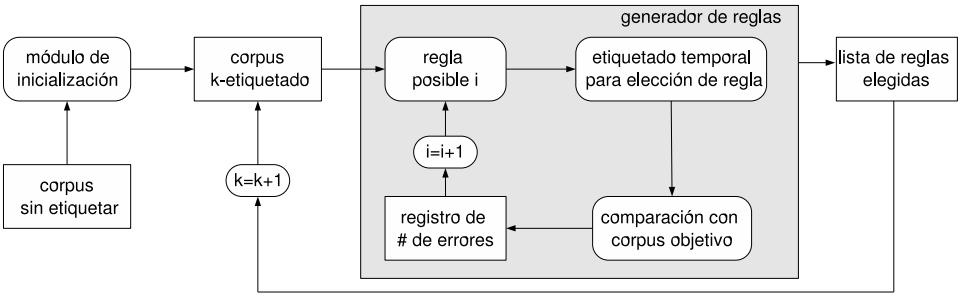

Figura 4.1: Algoritmo TBL

submódulo para generar reglas de contexto. A continuación se describe el algoritmo general y en la sección 4.4.2 se describe el módulo de aprendizaje del Brill POS Tagger.

4.4.1. Descripción del Algoritmo General de Aprendizaje de Brill (Algoritmo TBL)

Básicamente, este algoritmo de aprendizaje está basado en reglas o transformaciones y aprende del error (elige las mejores de entre un conjunto dado). La peculiaridad de este algoritmo es que el funcionamiento lingüístico de un idioma queda incorporado en las reglas aprendidas. De esta manera, una vez que se aprendieron las reglas, el usuario puede anexar las propias para mejorar la exactitud del etiquetador al que se incorporen.

Para poder aprender las reglas, el algoritmo de aprendizaje necesita 1) un corpus etiquetado manualmente, 2) ese mismo corpus pero sin etiquetas y 3) un conjunto de reglas de etiquetación pre-especificadas de donde se elegirán las mejores. Este conjunto de reglas pre-especificadas son realmente instancias de plantillas de la forma "cambiar etiqueta A por etiqueta B si *condición*". Es decir, el conjunto de reglas pre-especificado es realmente una combinatoria de transformaciones de etiquetas. El algoritmo de aprendizaje de Brill procede de la siguiente manera (ver figura 4.1). Primero, el texto sin etiquetar se introduce al *módulo de inicialización*, el cual etiquetará cada palabra del texto por primera vez con algún algoritmo[15]. (Es importante hacer notar que el algoritmo de aprendizaje de Brill está basado en transformaciones de etiquetas y por lo tanto necesita un etiquetado inicial.) Después, el texto etiquetado se introduce al módulo *generador de reglas*. La función de

[15]Esto es clave para la descripción del módulo de aprendizaje del POS Tagger de Brill como se verá más adelante.

este módulo es aprender la mejor regla posible (la que resulta en el menor número de errores al comparar con el corpus objetivo).

Para elegir una regla, el módulo generador de reglas procede en dos pasos que forman un loop: en la primera etapa examina cada posible regla (cada regla del conjunto de reglas pre-especificadas) y selecciona la que resulta en el menor número de errores al comparar con el corpus objetivo; en la segunda etapa re-etiqueta el corpus de acuerdo a la regla elegida en la primera, actualizando así el re-etiquetado de la iteración anterior. (Nota: La primera re-etiquetación se hace sobre el corpus etiquetado inicialmente en la etapa de inicialización con el módulo de inicialización.)

Esas dos etapas se repiten hasta lograr un criterio de terminación, tal como un mejoramiento insuficiente respecto a la iteración anterior.

En la figura 4.2 se muestra un ejemplo del algoritmo para el caso de un conjunto de transformaciones posibles de tamaño cuatro, tomado de [4]. Sea $\{T1, \ldots T4\}$ el conjunto de transformaciones para este ejemplo. El corpus sin etiquetar es etiquetado de forma inicial obteniendo así un corpus temporal con 5,100 errores en comparación al corpus objetivo. Después, se aplica cada posible transformación al corpus temporal (pero de forma temporal, es decir, no se guardan las modificaciones del corpus temporal) y se registra cada resultado. En el ejemplo, al aplicar $T2$ se obtiene el menor número de errores, entonces se aprende $T2$ y se guarda en la lista ordenada de reglas. Se aplica entonces $T2$ al corpus para así tener 3,145 errores. En este estado, $T3$ es la transformación que obtiene un menor número de errores, y entonces se aprende esta segunda regla. Después, en este estado, ya no es posible aprender nuevas reglas porque no se reduce el número de errores y entonces el aprendizaje termina con la lista ordenada de reglas: $T2$, $T3$.

Nótese que el cálculo del número de errores en la primera etapa requiere comparar la etiqueta (de cada palabra) lograda (con un candidato a regla) con la etiqueta correcta en el corpus objetivo (corpus manualmente etiquetado). En este sentido, el Algoritmo de Aprendizaje basado en Transformaciones es un algoritmo de aprendizaje supervisado, donde el feedback correctivo es el número de errores.

El output del algoritmo de Brill es una lista ordenada de transformaciones (reglas de etiquetado) que se convierte en un procedimiento de etiquetado y que puede ser aplicado a cualquier texto en el idioma del entrenamiento.

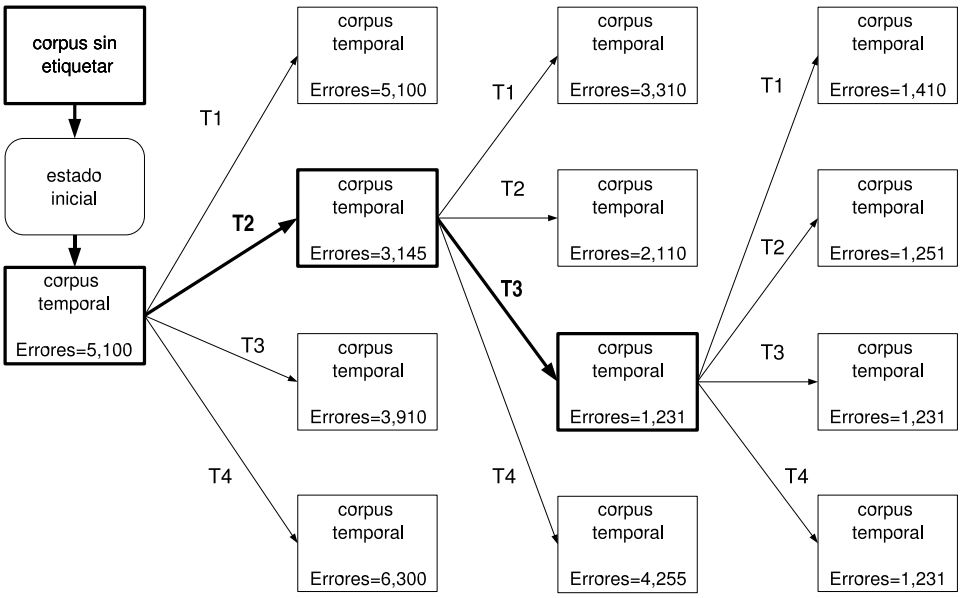

Figura 4.2: Algoritmo TBL

4.4.2. Módulo de entrenamiento del Etiquetador de Brill como Instancia del Algoritmo General

Ahora se describirá el módulo de entrenamiento del etiquetador de Brill. Para poder correr este módulo, primero hay que dividir al corpus manualmente etiquetado en dos partes de igual tamaño. Después, usando la primera mitad, hay que construir varios archivos (utilizando unas utilerías que Brill incluye en su aplicación): un lexicón y una lista de bigramas (pares de palabras que ocurren juntas con mucha frecuencia; por ejemplo, "Realmente era", "haría que") principalmente[16]. El lexicón que se construye con las utilerías es una lista de todos los tipos de palabras (types) del corpus de entrenamiento, organizada por renglones, dónde cada renglón tiene la forma

$$palabra\ etiqueta_1\ \ldots etiqueta_n$$

La $etiqueta_1$ es la etiqueta más probable de la *palabra*. Las restantes etiquetas, sin embargo, no siguen un orden de probabilidad. Recordemos aquí la distinción token/type: las etiquetas corresponden a ocurrencias (tokens) de la palabra (type) según su función en la oración.

[16]Y otros archivos secundarios. Para más detalles se recomienda ver la documentación que viene con el etiquetador de Brill.

El módulo de entrenamiento del etiquetador de Brill consta de dos submódulos: el léxico y el de contexto. Estos dos submódulos utilizan el algoritmo de aprendizaje basado en transformaciones descrito arriba para aprender dos conjuntos de reglas. Cada submódulo utiliza diferentes conjuntos de plantillas para generar las reglas (estos conjuntos de plantillas se describen en los cuadros 4.14 y 4.15). Dado que son instancias del algoritmo general, será suficiente con describir para cada uno su módulo de inicialización.

El submódulo léxico recibe como insumo el lexicón, la lista de bigramas (y otros parámetros), la primera mitad del corpus etiquetado y ese mismo corpus sin etiquetar (las plantillas generadoras de reglas están ya incorporadas en el módulo léxico).

La inicialización de etiquetas en este módulo léxico se ejecuta de la siguiente manera: A cada palabra en la primera mitad del corpus sin etiquetar se le asigna su $etiqueta_1$, si es que aparece en el lexicón; de otra manera, a las palabras desconocidas, se les asignará, como aproximación inicial, "NNP" si la palabra empieza con mayúscula y "NN" si no empieza con mayúscula[17]. (Esta etiquetación es la opción por default que trae el Brill Tagger, pero también permite intercambiarlas por las equivalentes del corpus usado. Por ejemplo, "NCMS000" en lugar de "NN" y "NP00000" en lugar de "NNP", que son las que yo usé.)

Con esta inicialización se entra al generador de reglas y al final se obtiene un output de reglas léxicas. En síntesis, el submódulo léxico aprenderá reglas para etiquetar palabras desconocidas (reglas léxicas) utilizando la primera mitad del corpus manualmente etiquetado.

El submódulo de contexto recibe como insumo el lexicón, los bigramas, la segunda mitad del corpus etiquetado, ese mismo corpus sin etiquetar y las reglas léxicas generadas por el submódulo léxico. En este submódulo la inicialización es igual que en el submódulo léxico, más la aplicación de las reglas léxicas generadas por el submódulo léxico a las palabras desconocidas.

Después de esta inicialización se entra al generador de reglas y se obtiene un conjunto de reglas de contexto. Como nota interesante añadamos que el submódulo de contexto utiliza el POS Tagger de Brill para etiquetar las palabras desconocidas aplicando las reglas léxicas. El POS Tagger de Brill se describe a continuación.

[17]NNP Proper Noun y NN Common Noun en el corpus Brown descrito arriba.

Cuadro 4.14: Plantillas de Reglas Léxicas

`x haspref(hassuf) l A`: si los primeros (últimos) l caracteres de la palabra son x, entonces se etiqueta a esta palabra con A.

`A x fhaspref(fhassuf) l B`: si la etiqueta de la palabra es A y los primeros (últimos) l caracteres de la palabra son x, entonces se etiqueta a esta palabra con B.

`x deletepref(deletesuf) l A`: si al borrarle a la palabra el prefijo (sufijo) x de longitud l se obtiene una palabra conocida (en el lexicón), entonces se etiqueta a esta palabra con A.

`A x fdeletepref(fdeletesuf) l B`: si la etiqueta de la palabra es A y si al borrarle a la palabra el prefijo (sufijo) x de longitud l se obtiene una palabra conocida, entonces se etiqueta a esta palabra con B.

`x addpref(addsuf) l A`: si al concatenarle a la palabra el prefijo (sufijo) x de longitud l se obtiene una palabra conocida (en el lexicón), entonces se etiqueta a esta palabra con A.

`A x faddpref(faddsuf) l B`: si la etiqueta de la palabra es A y si al concatenarle a la palabra el prefijo (sufijo) x de longitud l se obtiene una palabra conocida, entonces se etiqueta a esta palabra con B.

`w goodright(goodleft) A`: si la palabra se encuentra a la derecha de la palabra w entonces se etiqueta a esta palabra con A.

`A w fgoodright(fgoodleft) B`: si la etiqueta de la palabra es A y si se encuentra a la derecha (izquierda) de la palabra w, entonces se etiqueta a esta palabra con B.

`Z char A`: si la palabra contiene al carácter Z entonces se etiqueta a esta palabra con A.

`A Z fchar B`: si la etiqueta de la palabra es A y si contiene al carácter Z entonces se etiqueta a esta palabra con B.

Cuadro 4.15: Plantillas de Reglas de Contexto

`A B prevtag(nextag) C`: si la etiqueta de la palabra es A y si la etiqueta de la palabra anterior (siguiente) es C, entonces cambiar la etiqueta de la palabra a B.

`A B prev1or2tags(next1or2tags) C`: si la etiqueta de la palabra es A y si la etiqueta de alguna de las dos palabras anteriores (siguientes) es C, entonces cambiar la etiqueta de la palabra a B.

`A B prev1or2or3tags(next1or2or3tags) C`: si la etiqueta de la palabra es A y si la etiqueta de alguna de las tres palabras anteriores (siguientes) es C, entonces cambiar la etiqueta de la palabra a B.

`A B prev2tags(next2tags) C`: si la etiqueta de la palabra es A y si la etiqueta de alguna de la segunda palabra anterior (siguiente) es C, entonces cambiar la etiqueta de la palabra a B.

`A B prevbigram(nextbigram) C D`: si la etiqueta de la palabra es A y si la etiqueta de la segunda palabra anterior (palabra siguiente) es C y la de la palabra anterior (segunda palabra siguiente) es D, entonces cambiar la etiqueta de la palabra a B.

`A B surroundtag C D`: si la etiqueta de la palabra es A y si la etiqueta de la palabra anterior es C y la de la palabra siguiente es D, entonces cambiar la etiqueta de la palabra a B.

`A B curwd w`: si la etiqueta de la palabra es A y si la palabra es w, entonces cambiar la etiqueta de la palabra a B.

`A B prevwd(nextwd) w`: si la etiqueta de la palabra es A y si la palabra anterior (siguiente) es w, entonces cambiar la etiqueta de la palabra a B.

`A B prev1or2wd(next1or2wd) w`: si la etiqueta de la palabra es A y si alguna de las dos palabras anteriores (siguientes) es w, entonces cambiar la etiqueta de la palabra a B.

`A B prev2wd(nex2twd) w`: si la etiqueta de la palabra es A y si la segunda palabra anterior (siguiente) es w, entonces cambiar la etiqueta de la palabra a B.

`A B lbigram(rbigram) w x`: si la etiqueta de la palabra es A y si la palabra anterior (siguiente) es w, entonces cambiar la etiqueta de la palabra a B.

`A B wdand2bfr(rbigram) w(x) x(w)`: si la etiqueta de la palabra es A, si la palabra es x si la segunda palabra anterior (siguiente) es w, entonces cambiar la etiqueta de la palabra a B.

`A B wdprevtag(wdnexttag) C(w) w(C)` : si la palabra es w y tiene etiqueta A y, si la palabra anterior (siguiente) es w, entonces cambiar la etiqueta de la palabra a B.

`A B wdand2tagbfr(wdand2tagaft) C(w) w(C)` : si la palabra es w y tiene etiqueta A y si la etiqueta de la segunda palabra anterior (siguiente) es C entonces, cambiar la etiqueta de la palabra a B.

4.5. Funcionamiento del Etiquetador de Brill

Es conveniente insistir en la distinción entre el módulo de entrenamiento de Brill y el POS Tagger de Brill. El módulo de entrenamiento recibe un corpus manualmente etiquetado y aprende reglas para etiquetar textos nuevos. El POS Tagger de Brill utiliza las reglas generadas por el módulo de entrenamiento (o bien generadas de otra manera pero siguiendo las plantillas) para etiquetar textos nuevos. La figura 4.3 muestra el insumo y el producto del etiquetador de Brill.

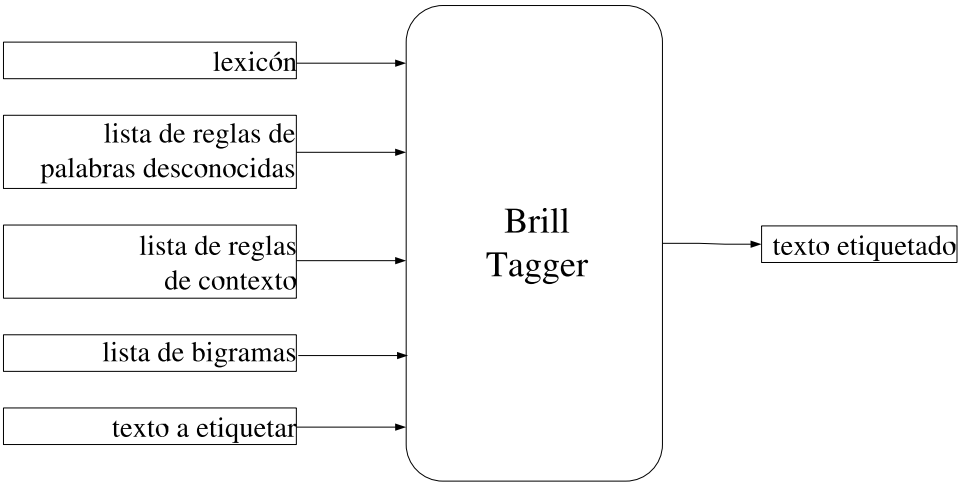

Figura 4.3: Input/Output del Etiquetador de Brill

Para correr[18], el etiquetador de Brill requiere recibir como input un lexicón (ver 4.16), una lista de reglas para etiquetar palabras desconocidas (ver 4.17), una lista de reglas de contexto (ver 4.18), una lista de bigramas (pares de palabras que ocurren juntas con mucha frecuencia; por ejemplo, "Realmente era", "haría que") que sirven para restringir la aplicación de las reglas léxicas generadas por cuatro plantillas (goodleft, fgoodleft, goodright, fgoodright ver cuadro 4.17) y el documento que se desea etiquetar. El proceso de etiquetado consta de tres fases en pipeline: búsqueda en el lexicón, aplicación de las reglas de palabras no en el lexicón, aplicación de las reglas de contexto.

En este módulo se aplican las reglas de contexto a todas las palabras del

[18]En la documentación de Brill Tagger, que viene junto con el programa, se dan las instrucciones para instalarlo y correrlo. Sólo corre en sistemas operativos de la familia Unix.

Cuadro 4.16: Extracto del lexicón que usa el etiquetador de Brill (y también el de Hepple)

Absolutely RB
Absorbed VBN
Abyss NN
Academically RB
Accepted JJ NNP
Accessories NNS NNP
Accomplishing VBG
According VBG
Accordingly RB
Accords NNPS NNP
Accountants NNPS

Cuadro 4.17: Extracto del conjunto de reglas léxicas que usa el etiquetador de Brill (pero no el de Hepple)

NN ed fhassuf 2 VBN x
NN ing fhassuf 3 VBG x
ly hassuf 2 RB x
ly addsuf 2 JJ x
NN $ fgoodright CD x
NN al fhassuf 2 JJ x
NN would fgoodright VB x
NN 0 fchar CD x
NN be fgoodright JJ x
NNS us fhassuf 2 JJ x

Cuadro 4.18: Extracto del conjunto de reglas de contexto que usa el etiquetador de Brill (y también el de Hepple)

NNS VBZ LBIGRAM , claims
NNS NNS CURWD prices
NNS VBZ NEXTBIGRAM CD NNS
NNS NNS PREV1OR2WD with
NNS NNS PREVWD back
NNS VBZ SURROUNDTAG NNP NNS
NNS VBZ NEXT2TAG POS
NNS VBZ LBIGRAM company plans
NNS VBZ PREVWD group
NNS NNS CURWD employees

texto (tanto a las desconocidas como a las conocidas).

Capítulo 5

Resolución del problema: VMP Tagger

Como ya se expresó antes, la idea original de esta tesis fue estudiar la arquitectura GATE con el objetivo de ejecutar tareas de extracción de información de textos en español. Después de estudiar el funcionamiento de GATE así como el de sus módulos, en cierto momento me di cuenta que el POS Tagger de Hepple no podía adaptarse para el español. Y esto porque el módulo Hepple no es open source y por lo tanto me fue imposible cambiar el algoritmo de etiquetado inicial. Este algoritmo etiqueta a las palabras desconocidas con "NN" y "NNP". "NNP" si la palabra empieza con mayúscula y "NN" si no empieza con mayúscula, según el sistema Penn Treebank. Como yo tenía que etiquetar según el sistema CLIC-TALP (el único que tuve disponible), era obligatorio que el Hepple me etiquetara en ese sistema.

Así pues, esta dificultad de no poder adaptar el Hepple para el español me impuso el subproblema de tener que buscar un módulo sustituto (o bien abandonar todo el proyecto de usar el GATE para extracción de información en español). La pista para la solución de este subproblema la encontré en el mismo módulo Hepple pues Hepple está basado en Brill (Brill-based POS Tagger). Hepple utiliza dos de las cuatro listas que genera el módulo de entrenamiento del Brill POS Tagger: el lexicón y las reglas de contexto. Como yo sabía desde que empecé a estudiar el Hepple que éste está basado en Brill, ya también tenía instalado en mi computadora el Brill POS Tagger y lo había probado, e incluso ya lo había entrenado para el español.

Como ya hemos dicho el módulo de entrenamiento del Brill POS Tagger genera cuatro listas (las reglas aprendidas) para la etiquetación. Y cuando

traté de meter las dos (de las cuatro) listas que utiliza el Hepple (lexicón y reglas de contexto) me dí cuenta que el Hepple no podía ser adaptado para el español. Pues cuando busqué el código fuente para cambiar las líneas de código que me generaban basura en el output me di cuenta que el código estaba encapsulado mediante un wrapper que llamaba a un paquete POS-Tagger.jar, es decir, clases de Java. Y estas clases eran ejecutables solamente (.class) y por lo tanto el código quedaba oculto.

En ese momento pedí ayuda a GATE (a su mailing list) donde me respondieron que no se podía ver el código porque no es open source. Por lo tanto decidí programar el módulo sustituto de Hepple, utilizando la misma idea que encontré en GATE de encapsularlo en un wrapper para poder acoplarlo a GATE pero también usarlo en forma independiente. Sin embargo, a diferencia de GATE que utilizó un wrapper para ocultar el código, yo use un wrapper como interfaz que permite el acoplamiento con GATE.

En resumen, mi aportación consiste en 1) la utilización del Brill POS Tagger para entrenamiento para el español con lo cual se salva un obstáculo para el procesamiento de lenguaje natural de textos en español utilizando GATE; 2) el desarrollo del módulo VMP Tagger que sustituye al módulo de GATE (Hepple) que es de código cerrado y en consecuencia no adaptable para el español. Estas dos aportaciones permiten ya ejecutar la tarea de extracción de información para el español después de adaptar otros módulos lo cual es un proyecto que continuaré en mi tesis de maestría.

5.1. Entrenamiento del Brill Tagger para el Español

Utilizando el corpus CLIC-TALP, se entrenó al etiquetador de Brill. El aprendizaje tardó aproximadamente 26 horas, y tuvo como resultado 4 archivos: un lexicón, una lista de reglas léxicas, una lista de reglas de contexto, y una lista de bigrama. En la documentación de la aplicación de Brill se encuentran las instrucciones para hacer que el Brill aprenda las reglas. Los pasos que se siguieron se resumen a continuación.

1. Se modificaron los programas unknown-lexical-learn.prl y start-state-tagger.c, para que se etiquetara con NP00000 (nombre propio) y NCMS000 (nombre común masculino singular) en lugar de NNP (proper noun) y NN (noun) respectivamente. El valor de la variable $THRESHOLD de los programas unknown-lexical-learn.prl y contextual-rule-learn se cambió a 1. Esta variable la utilizan ambos programas como criterio de

terminación del algoritmo de aprendizaje. El programa parará cuando el número de errores corregidos por las reglas (que son candidatas a ser elegidas por el módulo de aprendizaje) sea menor al $THRESHOLD[1].

2. Se modificó el formato (usando un programa en Python) del corpus de entrenamiento CLIC-TALP. Tenía el formato que a continuación se muestra (Brill lo necesita en el formato "palabra/etiqueta"):

```
La el DA0FS0
contemplación contemplación NCFS000
de de SPS00
la el DA0FS0
película película NCFS000
" " Fe
Howards_End howards_end NP00000
" " Fe
, , Fc
de de SPS00
James_Ivory james_ivory NP00000
```

3. Se creó un sólo archivo concatenando los 39 textos del CLIC-TALP.

4. Después, se dividió el corpus CLIC en 10 partes de igual tamaño utilizando el programa Utilities/divide-in-two-rand.prl que incluye Brill en su aplicación. Tal división la ejecuta eligiendo oración por oración en forma aleatoria. En realidad se modificó ese programa de perl para que dividiera el corpus en 10 partes y no en 2. Las primeras 9 partes se concatenaron para formar el corpus de entrenamiento, y la última parte se utilizó para evaluar los resultados.

5. Siguiendo las instrucciones de Brill, el corpus de entrenamiento se dividió en dos partes de igual tamaño. La primera, llamémosla TAGGED-CORPUS-1, fue usada para obtener las reglas para etiquetar palabras desconocidas; la segunda, llamémosla TAGGED-CORPUS-2, fue usada para aprender las reglas de contexto.

[1]En realidad, se puso a entrenar al Brill Tagger dos veces. La primera vez se utilizó el valor default de cada programa (3), pero no se obtuvieron resultados satisfactorios (porque no se aprendieron suficientes reglas). Por eso después se volvió a entrenar ahora con $THRESHOLD=1

6. El módulo que aprende las reglas que etiquetan a las palabras desconocidas utiliza el TAGGED-CORPUS-1 y el corpus total sin etiquetar (llamémoslo UNTAGGED-CORPUS-ENTIRE): es decir, las dos partes TAGGED-CORPUS-1 y TAGGED-CORPUS-2 concatenadas y sin etiquetas. Este corpus total se puede obtener utilizando el comando `cat` de Linux y el script /Utilities/tagged-to-untagged.prl que incluye Brill, como sigue:

   ```
   cat TAGGED-CORPUS-1 TAGGED-CORPUS-2 > TAGGED-CORPUS-ENTIRE
   cat TAGGED-CORPUS-ENTIRE | tagged-to-untagged.prl > \
   UNTAGGED-CORPUS-ENTIRE
   ```

7. Se tienen que crear algunos archivos antes de correr el programa de entrenamiento. Primero, la lista BIGWORDLIST que contiene a todas las palabras del corpus UNTAGGED-CORPUS-ENTIRE ordenadas por orden decreciente según su frecuencia. Para crear esta lista hay que correr en la consola el siguiente comando:

   ```
   cat UNTAGGED-CORPUS-ENTIRE | Utilities/wordlist-make.prl \
   | sort +1 -rn | awk '{ print $ 1 }' > BIGWORDLIST
   ```

8. Una segunda lista que se tiene que crear se denomina SMALLWORDTAGLIST y es de la forma (palabra etiqueta #veces), es decir, en cada renglón se indica el número de veces que la palabra se etiquetó con la etiqueta en el corpus TAGGED-CORPUS-1. Para crear esta lista se ejecuta el siguiente comando:

   ```
   cat TAGGED-CORPUS-1 | Utilities/word-tag-count.prl \
   | sort +2 -rn > SMALLWORDTAGLIST
   ```

9. La tercera y última lista, denominada BIGBIGRAMLIST, es una lista de todos los pares de palabras (bigramas) del corpus UNTAGGED-CORPUS-ENTIRE. Para crear esta lista se utiliza el siguiente comando:

   ```
    cat UNTAGGED-CORPUS-ENTIRE | \
   Utilities/bigram-generate.prl | \
   awk ' print 1,2' >BIGBIGRAMLIST
   ```

10. Después de crear las tres listas anteriores, se corre el módulo de aprendizaje de reglas para etiquetar palabras desconocidas como sigue:

    ```
    unknown-lexical-learn.prl BIGWORDLIST SMALLWORDTAGLIST \
    BIGBIGRAMLIST 500 LEXRULEOUTFILE
    ```

donde LEXRULEOUTFILE es el nombre del archivo donde se van a guardar las reglas que se vayan aprendiendo. El número 500 indica que sólo se usan los primero 500 renglones de la lista de bigramas. El número que Brill usó en sus instrucciones fue 300.

11. Después de unas 18 horas, el programa unknown-lexical-learn.prl terminó y se aprendieron 1034 reglas léxicas.

12. Para poder correr el módulo de aprendizaje de reglas de contexto se debe crear el archivo TRAINING.LEXICON como sigue:

```
cat TAGGED-CORPUS-1 | make-restricted-lexicon.prl > \
TRAINING.LEXICON
```

13. Después, se debe de crear el FINAL.LEXICON (que es el lexicón que se va a usar para etiquetar textos nuevos). Para crear este lexicón se utiliza todo el corpus etiquetado manualmente (TAGGED-CORPUS-ENTIRE):

```
cat TAGGED-CORPUS-ENTIRE | \
Utilities/make-restricted-lexicon.prl > \
FINAL.LEXICON
```

14. Después hay que desetiquetar la segunda mitad del corpus etiquetado con:

```
cat TAGGED-CORPUS-2 | tagged-to-untagged.prl > \
UNTAGGED-CORPUS-2
```

15. A continuación se etiqueta el corpus UNTAGGED-CORPUS-2 con el TRAINING.LEXICON y las reglas léxicas aprendidas LEXRULEOUTFILE con el comando:

```
tagger TRAINING.LEXICON UNTAGGED-CORPUS-2 BIGBIGRAMLIST \
LEXRULEOUTFILE /dev/null -w BIGWORDLIST -S > \
DUMMY-TAGGED-CORPUS
```

16. Por último, se corre el módulo de reglas de contexto de la siguiente manera:

```
contextual-rule-learn TAGGED-CORPUS-2 DUMMY-TAGGED-CORPUS \
CONTEXT-RULEFILE TRAINING.LEXICON
```

17. Después de unas 8 horas el programa contextual-rule-learn terminó y se aprendieron 488 reglas de contexto.

5.2. VMP Tagger: Módulo de Análisis Morfosintáctico para GATE

El VMP Tagger[2] está basado en el Brill Part-of-Speech Tagger y está programado en Java. Es adaptable a cualquier idioma, siempre y cuando se cuente con un corpus etiquetado manualmente. En cuyo caso se puede usar el módulo de entrenamiento del Brill Tagger para obtener cuatro listas: un lexicón, una lista de reglas léxicas, una lista de reglas de contexto y una lista de bigramas. VMP Tagger usa este output del módulo de entrenamiento del Brill para etiquetar textos nuevos (sin importar qué conjunto de etiquetas sea usado en el corpus de entrenamiento). Y tal etiquetación puede hacerse utilizando la arquitectura GATE o bien en forma independiente (ver figura 5.1).

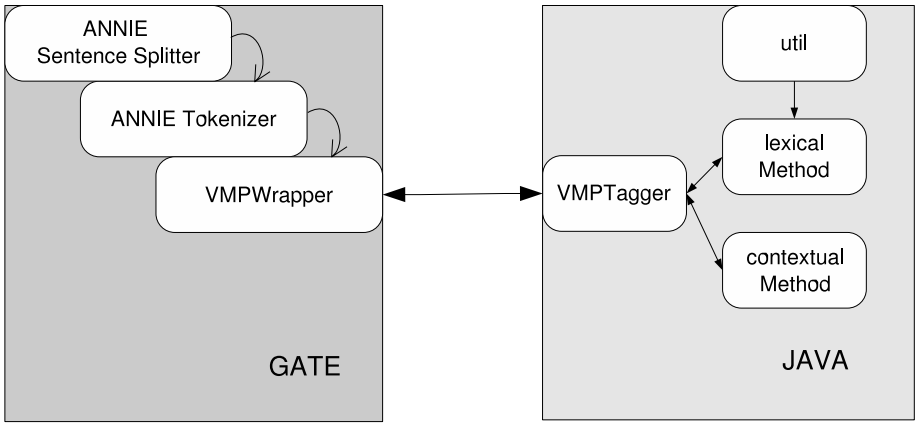

Figura 5.1: Arquitectura de VMP Tagger

Las características del VMP Tagger son:

1. Está basado en el Brill Tagger.
2. Es acoplable a GATE.
3. Es adaptable a cualquier conjunto de etiquetas.
4. Está programado en Java.
5. Es open source: disponible en http://sourceforge.net/projects/vmptagger.

[2]El código fuente se puede encontrar en http://sourceforge.net/projects/vmptagger

5.2.1. Arquitectura del Módulo VMP Tagger

Este etiquetador morfosintáctico está compuesto de las siguientes clases:

1. VMPWrapper.java
2. VMPTagger.java
3. contextualMethod.java
4. lexicalMethod.java
5. util.java

Estas clases se ilustran en el infograma de la figura 5.1. Para que el VMP Tagger se pueda correr exitosamente bajo GATE, primero necesitan correrse dos módulos de ANNIE: el tokenizador y el delimitador de oraciones (sentence splitter). El VMPWrapper sirve como enlace para acoplar el VMPTagger con GATE. Es la interfaz para el acoplamiento de VMPTagger con GATE. Nótese que el VMPTagger es independiente de GATE. (Para correrlo fuera de GATE el texto por etiquetar requiere estar "pretokenizado" –los símbolos de puntuación deben de estar precedidos de un espacio.)

El módulo VMPTagger usa el mismo algoritmo de etiquetado que el Brill POS Tagger. Es decir, primero etiqueta a las palabras con su etiqueta más probable si es que están en el lexicón (obtenido en el entrenamiento). En caso contrario se etiquetan con NP00000 (nombre propio) si empiezan con mayúscula o NCMS000 (nombre común masculino singular) si empiezan con minúscula. Después se aplican las reglas aprendidas en el entrenamiento del Brill: las reglas para etiquetar palabras desconocidas (palabras no en el lexicón) y las reglas de contexto. La aplicación de estas reglas corresponden a los submódulos lexicalMethod y contextualMethod respectivamente.

5.2.2. Instrucciones de Instalación del VMP Tagger en GATE

1. Bajar de http://sourceforge.net/projects/vmptagger la versión más nueva del VMP Tagger.
2. Descomprimir el paquete .tar.gz.
3. Correr GATE.

4. Hacer click en File "Load a CREOLE repository" y seleccionar el directorio donde se encuentra el contenido del archivo .tar.gz resultado de la descompresión. Ahora el módulo VMP Tagger será reconocido como módulo de GATE.

5. Cargar los siguientes recursos de procesamiento (processing resources) "ANNIE English Tokenizer", "ANNIE Sentence Splitter" y "VMP Tagger".

6. Cargar el recurso de lenguaje, es decir, el texto que se quiera analizar.

7. Crear una nueva pipeline y llamar los recursos de procesamiento del paso anterior, en el orden que fueron cargados (primero el tokenizador, luego el delimitador de oraciones, y al final en analizador morfológico).

8. Añadir el recurso de lenguaje a cada recurso de procesamiento.

9. Hacer click en "RUN"

5.2.3. Adaptación a otros idiomas (conjuntos de etiquetas)

1. Para cambiar el *encoding* que se usa al leer las listas de entrada (lexicón, reglas léxicas y de contexto, y bigrama) se necesita modificar el constructor de la clase VMPTagger.java (líneas 58 a la 61).

2. Para cambiar las etiquetas de etiquetado inicial (de "NN" y "NNP" a sus etiquetas equivalentes, según el corpus e idioma) modificar el método "runLexicon" de la clase VMPTagger.java (líneas 200 y 202)

5.2.4. Compilación y creación de paquete JAR

En caso de que se modifique el código fuente para adaptarlo a otro idioma, hay que volver a compilar las clases java y crear un paquete JAR. Para ello ejecutar en sucesión los siguientes comandos:

1. cd VMPTagger_V1.x/src

2. javac -classpath .:path_to_gate.jar:path_to_tools14.jar \
creole/vmptagger/VMPTagger.java

3. make the VMPTagger.jar file: jar cvf path_to_creole

5.2.5. Evaluación y Análisis de Errores de los etiquetadores Brill y VMP

Tomando una idea de Jurafsky [21], la evaluación consistió en comparar el 10 % del corpus, que se había reservado para evaluación, con ese mismo 10 % pero desetiquetado y vuelto a etiquetar con el Brill POS Tagger. Esta primera comparación evalúa la eficiencia del Brill POS Tagger y de las reglas aprendidas. Una segunda comparación compara el corpus de evaluación, con ese mismo corpus pero desetiquetado y vuelto a etiquetar con el VMPTagger. Esta segunda compración evalúa la eficiencia del VMP Tagger. La comparación con el corpus etiquetado por el Brill Tagger se ejecutó mediante un programa en Python. La comparación con el corpus etiquetado por el VMPTagger se realizó usando las herramientas para comparar textos incluidas en GATE. Los resultados fueron de un 93.85 % de exactitud (8812 correctas/ 9389 palabras en total) como evaluación del Brill y 93,79 % (8806 correctas / 9389 palabras en total) para el VMPTagger. A continuación se presenta la tabla de errores más comunes.

Cuadro 5.1: Tabla de errores más comunes. (El etiquetado humano es el correcto.)

etiquetado humano	etiquetado con Brill	num. errores	% errores
P0000000	P0300000	40	6,932409012
CS	PR0CN000	14	2,426343154
PP3CN000	P0300000	11	1,906412478
CS	PR0CN000	11	1,906412478
PR0CN000	CS	10	1,733102253
NCFP000	AQ0FP0	8	1,386481802

Cuadro 5.2: Comparación de los etiquetadores Brill, Hepple y VMP para el inglés. El Brill y el VMP usaron las listas que incluye Brill. (1) Se utilizó como input las listas que incluye el Hepple, (2) Se utilizó como input las listas que incluye Brill, (3) Se utilizó como input las listas de Hepple (lexicón y reglas de contexto) y dos archivos vacíos en lugar de las listas de bigrama y de reglas léxicas

texto	# pal.	Brill	Hepple(1)	Hepple(2)	VMP	VMP(3)
ce04	2326	94.71	93.42	95.23	95.53	95.4
cf05	2277	96.44	94.29	96.18	96.44	95.74

Capítulo 6

Conclusiones y Trabajo Futuro

Se ha presentado una descripción del problema de Extracción de Información dentro de la filosofía Open Source y de reutilización del software tomando como arquitectura de acoplamiento modular el entorno de desarrollo GATE. Además, dentro de esta perspectiva, se ha descrito la solución de un problema de interfaz para utilizar la arquitectura de GATE para el Procesamiento de Lenguaje Natural en español a través de un etiquetador morfosintáctico que tiene al menos el mismo desempeño que el etiquetador de Brill (el cual sólo corre en Unix).

Al entrar directamente al uso de una arquitectura compleja de software para realizar tareas de procesamiento de información, se puede percibir la gran complejidad que plantea un proyecto de ingeniería de software cuando el sistema a desarrollar excede las capacidades de una sola persona. Para lograr desarrollar un sistema de la complejidad de GATE se requiere (me parece) toda una organización.

Por eso para usuarios individuales y equipos pequeños de desarrollo de software, la solución está en las arquitecturas que permiten el desarrollo de componentes acoplables y/o y su reutilización. Y esto sólo puede lograrse dentro del modelo de desarrollo de la Open Source Initiative que da al software una oportunidad para crecer y desarrollarse.

Con esta tesis creo haber dado el primer paso para mi aportación futura en esa dirección abierta. Espero que el VMP Tagger pueda crecer y mejorar con la colaboración de desarrolladores de software en el campo del procesamiento de lenguaje natural.

Como trabajo futuro es necesario para hacer extracción de información

para el español me parece necesario:

1. Etiquetar más corpus para el español. Como punto de comparación digamos que el corpus Brown (un corpus para el idioma inglés) consta de aproximadamente un millón de palabras mientras que el CLIC-TALP (el corpus para el español que yo usé) tiene solamente 100,000.

2. Adaptar para el español los otros módulos de GATE que se necesitan para la tarea de extracción de información (como se comento en el capítulo 3).

3. Analizar los errores de etiquetado en las reglas generadas por el módulo de aprendizaje del Brill Tagger y agregar manualmente nuevas reglas utilizando expertos humanos en lingüística.

4. Mejorar la eficiencia del VMP Tagger utilizando autómatas finitos (cf. [26]).

Bibliografía

[1] J. A. Alonso, T. Badia, y J. G. Arroyo. *Tecnologías del Lenguaje*. UOC, 2003.

[2] M. C. Bettoni. Kant and the software crisis: suggestions for the construction of human-centred software systems. *AI Soc.*, 9(4):396–401, 1995. ISSN 0951-5666.

[3] T. Brants. Tnt – a statistical part-of-speech tagger, 2000.

[4] E. Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–565, 1995.

[5] M. Civit. Guía para la anotación morfológica del corpus clic-talp, 2002.

[6] H. Cunningham. *Software Architecture for Language Engineering*. Tesis Doctoral, University of Sheffield, 2000.

[7] H. Cunningham, K. Humphreys, R. Gaizauskas, y M. Stower. Creole developer's manual. Informe técnico, Department of Computer Science, University of Sheffield, 1996. `http://gate.ac.uk/`.

[8] H. Cunningham, D. Maynard, K. Bontcheva, y V. Tablan. Gate: A framework and graphical development environment for robust nlp tools and applications. En *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*. 2002.

[9] S. J. DeRose. Grammatical category disambiguation by statistical optimization. *Comput. Linguist.*, 14(1):31–39, 1988. ISSN 0891-2017.

[10] J. Feller y B. Fitzgerald. A framework analysis of the open source software development paradigm. En *ICIS '00: Proceedings of the twenty first international conference on Information systems*, páginas 58–69.

Association for Information Systems, Atlanta, GA, USA, 2000. ISBN ICIS2000-X.

[11] R. Gaizauskas, P. Rodgers, H. Cunningham, y K. Humphreys. Gate user guide, 1996. `http://gate.ac.uk/`.

[12] W. M. Gentleman. If software quality is a perception, how do we measure it? En *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*, páginas 32–43. Chapman & Hall, Ltd., London, UK, UK, 1997. ISBN 0-412-80530-8.

[13] T. J. Halloran y W. L. Scherlis. High quality and open source software practices. 2002.

[14] L. Hatton. Linux and the cmm. 1999.

[15] M. Hepple. Independence and commitment: Assumptions for rapid training and execution of rule-based POS taggers. En *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics (ACL-2000)*. 2000.

[16] J. E. Hopcroft y J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.*. Addison-Wesley, 1979. ISBN 0-201-02988-X.

[17] Humphrey. Comments on software quality. En *Distributed to the National Conference of Commissioners on Uniform State Laws for their Annual Meeting Sacramento, CA*. 1997.

[18] W. S. Humphrey. A personal commitment to software quality. 1995.

[19] *Software engineering – Product quality – Part 1: Quality model*, 2001. ISO/IEC 9126-1:2001.

[20] I. JavaSoft, Sun Microsystems. *JDK 5.0 Documentation*. Sun Microsystems, Inc., 2004.

[21] D. Jurafsky y J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, 2000. ISBN 0130950696.

[22] S. H. Kan. *Metrics and Models in Software Quality Engineering*. ISBN: 0201729156. Pearson Education, 2 edición, 2002.

[23] S. Koch y G. Schneider. Results from software engineering research into open source development projects using public data. Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, H.R. Hansen und W.H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversität Wien, 2000.

[24] M. P. Marcus, B. Santorini, y M. A. Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1994.

[25] E. S. Raymond. *The Cathedral and the Bazaar*. Thyrsus Enterprises, 3.0 edición, 2000.

[26] E. Roche y Y. Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, 1995.

[27] D. C. Schmidt y A. Porter. Leveraging open-source communities to improve the quality & performance of open-source software. Submitted to the First Workshop on Open-Source Software Engineering, 2001.

[28] D. A. Watt. *Programming Language Design Concepts*. John Wiley & Sons, Ltd, 2004.