Una arquitectura software para el desarrollo de aplicaciones de generación de lenguaje natural

Carlos García Ibáñez
Facultad de Informática,
Universidad Complutense de Madrid
28040 Madrid
cgarcia@lucent.com

Raquel Hervás Ballesteros
Facultad de Informática,
Universidad Complutense de Madrid
28040 Madrid
lauranar@yahoo.com

Pablo Gervás
Facultad de Informática,
Universidad Complutense de Madrid
28040 Madrid
pgervas@sip.ucm.es

Resumen: Se plantea el desarrollo de un *framework* orientado a objetos para Sistemas de Generación de Lenguaje Natural (GLN). Se revisan las arquitecturas utilizadas hasta la fecha, se revisa una propuesta de arquitectura genérica, y se describe el tipo de solución que se aspira a conseguir. A partir de esa información se muestra cómo la utilización de metodologías típicas de Ingeniería de Software, como la Orientación a Objeto o los Patrones de Diseño, resuelve algunos de los problemas más acuciantes de los desarrolladores de aplicaciones de GLN. Se presenta el *framework* implementado y se plantean las aplicaciones en desarrollo que se basan en el código presentado.
Palabras clave: Generación De Lenguaje Natural, Reutilización de Software, Patrones de Diseño

Abstract: This paper proposes the development of an object oriented framework for Natural Language Generation Systems (NLG). It reviews the architectures used to date and a proposal for a generic architecture, and it describes the type of solution that is desired. From this information it is shown how the use of software engineering techniques, like Object Oriented Design or Design Patterns, can solve the more pressing problems of NLG. The implemented prototype is described together with two applications under development which use the framework.
Keywords: Natural Language Generation, Software Reuse, Design Patterns

1 Introducción

En los últimos años, con la llegada de los ordenadores personales al gran público, la interacción hombre-máquina ha ido evolucionando hasta convertirse en un elemento fundamental de cualquier sistema informático. Si un sistema está preparado para comunicar la información que debe presentar al usuario en lenguaje natural, este será más receptivo a la misma y tendrá más confianza en la aplicación.

Hay diversos métodos a la hora de hacer que una aplicación genere mensajes en lenguaje natural, con distintos grados de complejidad y flexibilidad. En un extremo del espectro, tenemos los tradicionales "textos enlatados" ("canned texts"): ante determinadas situaciones, el sistema muestra mensajes cableados directamente en el código del programa. Esta solución es la menos flexible y más dependiente del dominio de aplicación, pero proporciona resultados bastante aceptables para sistemas sencillos. En el otro extremo, tenemos la generación de lenguaje natural basada en conocimiento ("deep generation" o generación profunda): el sistema genera todos los mensajes de forma dinámica, basándose en el conocimiento lingüístico y el conocimiento del mundo en forma de ontología o jerarquía conceptual de que dispone. El comportamiento del sistema es mucho más flexible, adaptable y ampliable. Algunos sistemas actuales utilizan soluciones híbridas, enlazando texto enlatado o

plantillas ("templates"; fragmentos de texto que se utilizan repetidamente en distintos documentos del mismo dominio en los que varían sólo algunos de sus elementos, por lo que pueden ser fácilmente parametrizables para adaptarlos a distintas situaciones) con fragmentos generados de forma dinámica a partir de una ontología propia del dominio.

El problema fundamental de los generadores actuales es que su construcción es compleja y está muy ligada al dominio en el que se aplican, con lo que la reutilización del software es escasa o prácticamente nula. Tener que moverse en un espectro tan amplio de posibilidades de diseño como el descrito hace que esta característica del campo sea especialmente grave.

En otras áreas de la Informática se viene utilizando desde hace algún tiempo la idea de *framework*. Un *framework* define un modelo de proceso y de datos común a todas las aplicaciones informáticas de un mismo dominio. No se trata de una aplicación completa, sino que deja huecos que tienen que ser rellenados por el desarrollador del sistema concreto. Así, tenemos *frameworks* para aplicaciones gráficas, ofimáticas, educativas, etc.

Nuestra intención es seguir una línea similar para aplicaciones de generación automática de lenguaje natural, de forma que a la hora de diseñar e implementar un nuevo sistema de GLN se pueda reutilizar el trabajo previamente desarrollado. Para ello combinamos trabajo previo sobre definición de arquitecturas genéricas para el desarrollo de aplicaciones de generación de lenguaje natural (Reiter & Dale, 2000; Cahill et al, 2001), con ideas de patrones de diseño (Gamma et al, 1995) y frameworks (Johnson & Foote,1988).

2 Fuentes para la especificación de requisitos de un framework GLN

Tenemos en cuenta tres fuentes fundamentales para establecer las restricciones de arquitectura que debe satisfacer un *framework* para desarrollar aplicaciones de GLN: el trabajo teórico existente sobre arquitecturas de sistemas GLN, una propuesta de arquitectura genérica fruto de un trabajo explícito de revisión del estado del arte en el campo, y las ideas básicas sobre patrones de diseño y frameworks.

2.1 Arquitecturas de sistemas GLN

Los sistemas de GLN que se han venido desarrollando hasta ahora presentan distintas arquitecturas (DeSmedt, 1996), aunque todas tienen en común una organización modular de las tareas.

Las tareas fundamentales que se contemplan son determinación de contenido - que se va a decir -, planificación del documento - cómo se va a organizar -, generación de expresiones de referencia - cómo describir cada objeto -, agregación - cómo agrupar lo que se dice -, lexicalización - qué término concreto usar en cada caso -, y realización superficial - cómo dar forma lingüística al contenido resultante.

La forma en que se comunican estos módulos es la que define la arquitectura del sistema. Así, nos encontramos con las siguientes variantes:

- **Integrada** o **monolítica**: el sistema es un solo bloque en el que no hay una división clara entre los módulos (Kantrowitz, 1992). Suele resultar bastante eficiente, pero se impide por completo la reutilización de partes específicas del sistema.
- **Secuencial** (*pipeline*): los módulos se encadenan en un flujo de información unidireccional (Reiter & Dale , 2000). Su principal desventaja es que las decisiones tomadas en una fase deben mantenerse a lo largo de toda la cadena, sin posibilidad de revisión o mejora.
- **Interactiva** (*retroalimentación* o *feedback*): se permite una revisón de las decisiones tomadas. La información circula adelante y atrás en el sistema: la salida de un módulo no tiene por qué ser completa, y revisarse en función de decisiones posteriores (Hovy, 1988).
- **De pizarra** (*oportunista*): la información no viaja entre los distintos módulos hasta completar su tratamiento, sino que está en una zona común, accesible desde todos ellos, de forma que puedan manipularla de manera simultánea hasta que se obtiene el resultado (Calder et al, 1999).
- **Basada en revisión**: la información fluye de forma cíclica entre los módulos hasta que el resultado alcanza un estado óptimo (Robin, 1994).

Prácticamente los módulos son los mismos en todos los sistemas, variando únicamente la

forma en la que están interconectados. Así, en un *framework*, habría que incluir los elementos funcionales y una estructura elemental que permitiera distintas opciones de intercomunicarlos, dejando a la elección del desarrollador su configuración final.

2.2 La arquitectura genérica RAGS

Tras un análisis bastante exhaustivo de las arquitecturas de sistemas GLN existentes, el proyecto RAGS (Cahill et al, 2001) propone un conjunto de estructuras de datos para representar la información con la que deben trabajar los distintos módulos de un sistema de GLN, así como las operaciones que se pueden realizar sobre dichas estructuras, de manera suficientemente genérica para dar cabida a las distintas teorías lingüísticas existentes. Sin embargo, evitan definir el flujo de control y de datos del sistema, dejando esta decisión al criterio del implementador del mismo.

En vista de la revisión realizada, la arquitectura RAGS resume los niveles básicos de representación necesarios para un sistema GLN:

- **Nivel Conceptual**: conocimiento no lingüístico, generado y gestionado por la aplicación que hace uso del sistema de GLN. Por supuesto, es completamente dependiente del dominio de la aplicación.
- **Nivel Semántico**: representación del significado desde el punto de vista lingüístico.
- **Nivel Retórico**: organización de las estructuras guiada por la retórica, para dotar de coherencia al texto (Mann & Thompson, 1988).
- **Nivel de Documento**: estructuración del discurso en una jerarquía de funciones textuales, i.e. subdivisión en epígrafes, capítulos, párrafos...
- **Nivel Sintáctico**: comprende el procesamiento sintáctico de los elementos generados.

Estos niveles pueden ir apareciendo en orden en un sistema de GLN. Por ejemplo, el nivel conceptual debe ser el primero en ser procesado, mientras que la salida final debe estar compuesta de estructuras sintácticas y de documento. Sin embargo, si queremos proporcionar una cierta flexibilidad a la hora de decidir la estructura del sistema, no podemos forzar la inclusión de todos los niveles ni fijar el orden en que se encadenan.

2.3 Patrones de diseño y frameworks

Los patrones de diseño son una serie de "recetas" extraídas de la experiencia de la propia comunidad de desarrolladores, y que a lo largo del tiempo se han mostrado como buenas soluciones a distintos problemas que aparecen de manera recurrente a la hora de plantearse el diseño de una aplicación informática (Gamma et al., 1995).

Un framework ofrece un conjunto de clases e interfaces interrelacionadas en forma de un diseño reutilizable para una familia de sistemas, con una fuerte cohesión estructural (jerarquía de clases, herencia y composición) y de comportamiento (modelo de interacción de los objetos del framework) (Johnson & Foote,1988). Cuando diseñamos una aplicación basándonos en un framework, decimos que estamos implementando una *instancia* del mismo, es decir, estamos *rellenando los huecos* que le faltan al framework para ser una aplicación concreta. El framework soporta la estructura general de cualquier aplicación del dominio al que se adscribe, a modo de esqueleto, y son esos huecos los que aportan la flexibilidad requerida para ajustar la aplicación a nuestros intereses concretos.

Durante su desarrollo, el framework pasa por distintas fases en las que se va refinando su funcionalidad y los servicios que proporciona a los clientes. En un principio el usuario del framework tiene que construir las implementaciones concretas de los componentes de la aplicación, cuya interfaz o diseño abstracto facilita el framework. En esta fase, el framework se limita a ofrecer una plantilla genérica para el sistema, y es el usuario el que tiene que implementar el comportamiento interno de cada uno de los componentes. Estos son los llamados "frameworks de caja blanca". Las clases que el usuario va generando durante este uso del framework pueden ir incorporándose a la biblioteca del clases del mismo.

Con el tiempo, el framework se va convirtiendo en una "caja negra": la biblioteca de clases se va poblando con realizaciones concretas de los componentes abstractos, disponibles directamente para el usuario y cuya implementación él no tiene por qué conocer. Para el usuario es mucho más sencillo trabajar con este tipo de frameworks, puesto que basta

con que sea consciente de las entidades de las que dispone y cuál es su comportamiento a alto nivel. Además de esta ventaja, el usuario mantiene la posibilidad de ampliar y mejorar esas entidades igual que en un framework de caja blanca.

3 FROGS

FROGS (Framework-like RAGS Oriented Generation System) nace como un intento de aplicar las técnicas actuales de industrialización del software al campo de la GLN, basándose en las ideas propias del dominio y en los avances realizados en el mismo en cuanto a arquitecturas estándar. FROGS no pretende introducir ideas nuevas en el dominio, sino simplemente enfocar las ya existentes desde el punto de vista de los frameworks y los patrones de diseño.

Esta propuesta pretende proporcionar a un posible diseñador de aplicaciones GLN la infraestructura necesaria para facilitar su labor al máximo, por medio de los esquemas y estructuras arquitectónicas genéricas usados comúnmente en este tipo de sistemas.

Inicialmente FROGS está pensado como un framework de caja blanca: proporcionaría una base arquitectónica genérica para distintos tipos de sistemas GLN, con soporte para algunas estructuras de datos estándar, dejando al usuario la posibilidad de definir por completo las distintas etapas de su sistema. Sin embargo, la aspiración de FROGS es la de llegar a convertirse en un framework de pleno derecho en el dominio de la GLN.

Con objeto de garantizar que las ideas básicas sobre las que se construye la arquitectura no son dependientes de lenguajes de programación específicos se ha trabajado en paralelo en versiones de la arquitectura en Java (por su portabilidad) y C++ (por su eficiencia).

En este apartado se describe el diseño de FROGS, explicando algunas que las decisiones que se han tomado al llevarlo a cabo.

3.1 Pilares de una arquitectura GLN

Para describir la arquitectura de cualquier sistema de GLN, es necesario definir tres aspectos fundamentales:

- Conjunto de etapas o módulos que componen el sistema. Cada una de estas etapas tiene un funcionalidad concreta: procesa los datos que recibe como entrada y devuelve una salida que puede servir como entrada para otro módulo a su vez o puede ser la salida final del sistema, en forma de texto, por ejemplo.
- Flujo de control que lo gobierna, encargado de decidir cómo se conectan los módulos y cómo se transfieren los datos de unos a otros. Hay distintas opciones posibles dependiendo de las necesidades del sistema: arquitecturas lineales, cíclicas, de pizarra, o cualquier combinación de ellas.
- Estructuras de datos específicas que se intercambian los módulos.

Idealmente estos puntos se comportarían como ejes independientes, de manera que la modificación de sólo uno de ellos daría lugar a una arquitectura diferente. Por ejemplo, dado un conjunto de módulos y las estructuras de datos que circulan entre ellos, si se varía únicamente el flujo de control tendríamos la posibilidad de definir arquitecturas lineales, iterativas o incluso de pizarra, con un esfuerzo mínimo.

Estos tres ejes no son completamente ortogonales, y existen dependencias entre ellos. Uno de nuestros objetivos será encontrar un diseño que reduzca este acoplamiento.

3.2 Las etapas del sistema: `Stage` y `StageSet`

El primer paso es proporcionar una definición genérica del concepto de módulo o etapa de un sistema GLN. La naturaleza de estas etapas es muy diferente dependiendo del nivel lingüístico del que se ocupen o de los tipos de datos que manejen, pero todas tienen en común un comportamiento procedimental: al ejecutarse, generan una salida que es el resultado de procesar unos datos de entrada. Es decir, una etapa genérica debe: recibir datos de entrada, ejecutar el procesamiento de los datos, y devolver el resultado.

Los datos de entrada y los de salida deben tener el mismo formato, puesto que vamos a conectar unas etapas con otras, así que las salidas tienen que ser compatibles con las entradas. Esto se consigue mediante una estructura genérica `Draft`, cuyas características se detallan más adelante.

Con objeto de mantener la generalidad, no debe presuponerse nada sobre la manera de procesar los datos. La implementación del comportamiento interno de cada módulo se deja como responsabilidad del usuario de FROGS, el constructor del sistema GLN final.

A partir de esta información se define la interfaz de una etapa (`Stage`), cuyo único

método método, `run`, recibe un elemento `Draft` como entrada, y, como resultado de procesarlo, devuelve otro `Draft`.

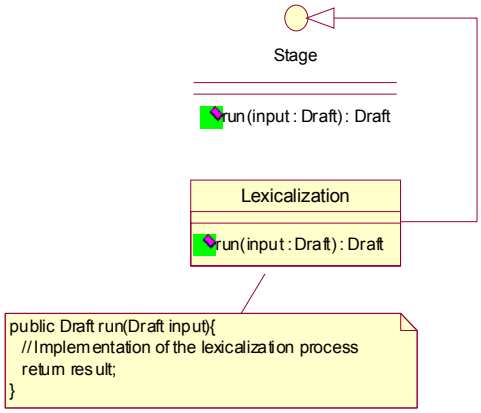

Figura 1 Interfaz Stage

De esta manera, el usuario que desee crear una etapa de Lexicalización simplemente tendrá que implementar la interfaz `Stage`, definiendo del proceso dentro del método `run` (figura 1).

FROGS proporciona algunas implementaciones estándar de estos módulos, como un Lexicalizador, un Planificador de Contenidos y un Realizador Superficial (`Lexicalization`, `ContentDetermination` y `SurfaceRealization`, respectivamente), que se agrupan en un `StageSet` al que se puede acceder a través de una factoría abstracta (Gamma et al, 1995) a la que llamamos `StageSetFactory`.

Para cada una de esas etapas, se facilitan distintas versiones que se diferencian en las estrategias utilizadas. Dadas las dependencias entre etapas que puede ocasionar el uso de estas tecnologías concretas, FROGS también ofrece la posibilidad de utilizar conjuntos predefinidos de etapas, a modo de factorías de objetos – realizaciones concretas del `StageSetFactory` abstracto – para obtener familias completas de módulos, de manera que esas dependencias se ocultan al usuario del framework.

3.3 El Control de Flujo: `ControlFlow`

La misión del Control de Flujo (`ControlFlow`) es la de decidir en qué orden se disponen y ejecutan las etapas del `StageSet`, si es que existe alguno o si alguna debe ejecutarse más de una vez. El objetivo es hacer el control de flujo lo más independiente posible de las etapas concretas que maneja, así que debe tratar con ellas únicamente a nivel de interfaz, no de implementación.

Por otro lado, las clases clientes de `ControlFlow` (las que hacen uso de él), no tienen que conocer ningún detalle de su topología interna, así que define una interfaz para acceder a las etapas del `StageSet` de manera aparentemente secuencial, aunque dependiendo de la implementación concreta seleccionada tendremos un recorrido lineal, basado en revisión o en estrategias oportunistas, por ejemplo. Para ocultar estos detalles de implementación, seguiremos las pautas del patrón *Iterator*.

Para hacer esto, `ControlFlow` dispone de un método `getNextStage`, que devuelve la siguiente etapa que debe ser ejecutada, y un método `end` que devuelve cierto cuando no queda ninguna etapa por ejecutarse.

Por supuesto, `ControlFlow` debe conocer el conjunto de etapas (`StageSet`) sobre las que tiene que actuar . Esta información se introduce a través del método `setStageSet`, al que pasamos el `StageSet` que nos interesa.

Actualmente FROGS implementa versiones básicas de los flujos de control lineal, con revisión, y oportunista, aunque el cliente de FROGS puede definir sus propios flujos de control, siguiendo la interfaz descrita.

3.4 Las Estructuras de Datos: `Draft`

La entrada y salida de datos de cada etapa está definida por medio de una estructura `Draft`, que sirve de envoltorio para datos de todos los niveles que conviven usualmente en un sistema GLN: representaciones semánticas, retóricas, de documento, sintácticas y textuales. Estas representaciones siguen las interfaces definidas por RAGS, de manera que el usuario del framework pueda instanciar `Draft` con las implementaciones RAGS que prefiera. FROGS proporciona una implementación básica de Draft (`DraftImpl`) que utiliza objetos que cumplen esas interfaces.

3.5 Atando cabos: `Architecture`

Por último, la clase `Architecture` define, a modo de plantilla (*Template Method*)(Gamma et al, 1995) los pasos a seguir para ejecutar cada una de las etapas definidas en su `StageSet`, iterando secuencialmente sobre ellas según le

indica su `ControlFlow`. Así, esta clase no tendría que ser redefinida, simplemente instanciada por el cliente del framework, seleccionando el `StageSet` y el `ControlFlow` que prefiera.

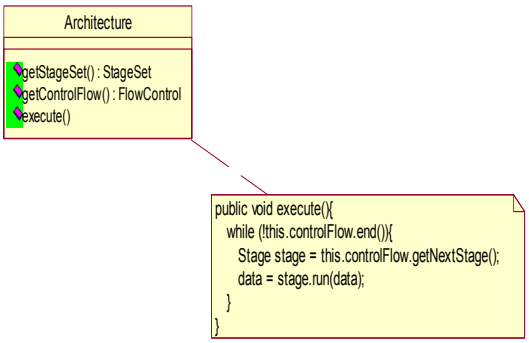

Figura 2 Clase Architecture

Para construir una aplicación con el conjunto de módulos por defecto y control de flujo lineal bastaría con escribir la línea:

```
Architecture jGoLeN_1 =
 New Architecture(new DefaultStageSet(),
                  new LinearControlFlow());
```

Mientras que para obtener un conjunto de módulos específico y control de flujo con revisión habría que hacer:

```
Architecture jGoLeN_2 =
 New Architecture(new jFUFStageSet(),
                  new RevisionsControlFlow());
```

4 PIPELINEWEATHER

Para demostrar la flexibilidad de la biblioteca y su facilidad de uso, se está desarrollando con la misma una aplicación sobre meteorología. Lo que se describe a continuación son los módulos que como usuarios de cFrogs hemos implementado para esta aplicación concreta.

4.1 Ejemplo de uso

Ésta es una versión simple de arquitectura de *pipeline* para el tratamiento de ciertos hechos sobre meteorología (figura 3). Ahora tenemos una clase `Application` externa a la biblioteca, y que es la que se encarga de leer y escribir los datos de entrada y salida en un fichero.

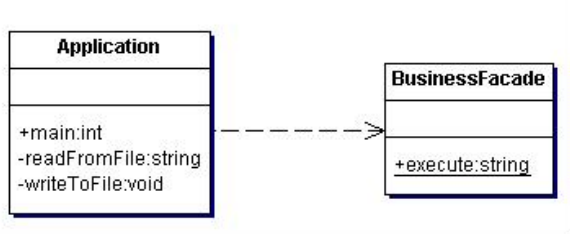

Figura 3 Conexión con la aplicación

En la clase `BusinessFacade` se encuentra la interfaz de comunicación con la aplicación, siguiendo el patrón de diseño *Facade* (Gamma et al, 1995).

En este ejemplo se utilizan etapas de `ContentDetermination`, `DiscoursePlanning`, `Lexicalization` y `SurfaceRealization`, gestionadas por un `StageSet`, y utilizando un flujo de control secuencial `LinearControlFlow`.

El fichero con los datos de entrada tendrá la siguiente estructura:

```
temperature(Madrid,15,today)
cloudiness(Madrid,5,today)
temperature(Madrid,20,tomorrow)
cloudiness(Barcelona,7,today)
cloudiness(Bilbao,10,tomorrow)
```

Los distintos hechos están separados por finales de línea. El contenido de este fichero es leído por `Application` y pasado a `BusinessFacade` en una cadena de caracteres.

Antes de poder ejecutar el flujo de control especificado es necesario transformar la cadena de caracteres de entrada en la estructura con la que trabajan los módulos: un `Draft`. De esta tarea se encarga un módulo específico, similar a las etapas internas del sistema, pero que no es gestionado directamente por el flujo de control. Se trata de un preprocesador de los datos de entrada, y deberá modificarse cada vez que se cambie el formato de trabajo.

El resultado de este preproceso es una lista de representaciones semánticas, cada una representando uno de los hechos de entrada.

```
-----CONCEPTUAL REPRESENTATION-------

 Input (facts):

temperature(Madrid,15,today)
cloudiness(Madrid,5,today)
temperature(Madrid,20,tomorrow)
cloudiness(Barcelona,7,today)
cloudiness(Bilbao,10,tomorrow)
```

```
 Output (SemRep list):

temperature(Madrid,15,today)
cloudiness(Madrid,5,today)
temperature(Madrid,20,tomorrow)
cloudiness(Barcelona,7,today)
cloudiness(Bilbao,10,tomorrow)
```

En el flujo de control la primera etapa se ocupa de identificar el mensaje a transmitir. En esta fase se filtran los hechos por ciudad y día, quedándonos en este ejemplo con los que tienen por valores "Madrid" y "today".

```
-------CONTENT DETERMINATION--------

 Input predicates(SemRep list):

temperature(Madrid,15,today)
cloudiness(Madrid,5,today)
temperature(Madrid,20,tomorrow)
cloudiness(Barcelona,7,today)
cloudiness(Bilbao,10,tomorrow)

 Output predicates(SemRep list):

temperature(Madrid,15,today)
cloudiness(Madrid,5,today)
```

La siguiente etapa de este sistema, se ocupa de organizar el contenido del mensaje para su realización. Aquí se transforma la lista de represenataciones semánticas en una representación retórica, separando los hechos en nuevas representaciones semánticas que quedarán en las hojas de la estructura retórica de árbol.

```
---------DISCOURSE PLANNING----------

 Input predicates(SemRep list):

temperature(Madrid,15,today)
cloudiness(Madrid,5,today)

 Output rhetorical representation:

description[localization(Madrid,today),
conjunction[temperature(15),cloudiness(5)]]
```

En la siguiente etapa, mediante el uso de plantillas se transforman las representaciones semánticas de la estructura retórica en una estructura de documento, que especifica cómo se van a presentar tipográficamente.

```
---------LEXICALIZATION----------

 Input rhetorical representation:

description[localization(Madrid,today),
           conjunction[temperature(15),
                       cloudiness(5)]]

 Output documental representation:

the weather in Madrid for today
                       is the following
the temperature is 15 degree centigrade
it is cloudy
```

La última fase se encarga de concatenar las cadenas de caracteres que contiene la estructura de documento según las relaciones entre las hojas del árbol de la estructura retórica. Además se pone en mayúscula la primera letra de la primera cadena y se añade un punto al final.

```
---------SURFACE REALIZATION----------

 Input documental representation:

the weather in Madrid for today
                       is the following
the temperature is 15 degree centigrade
it is cloudy

 Final output:

The weather in Madrid for today is the
following: the temperature is 15 degree
centigrade and it is cloudy.
```

Aunque este ejemplo es extremadamente simple, ilustra la facilidad con que módulos muy simples pueden interconectarse con poco esfuerzo de programación para conseguir resultados de GLN aceptables.

5 Cuestiones a resolver en un framework para aplicaciones GLN

Al plantear la definición de un *framework* de estas características, se han identificado una serie de dificultades que debería resolver el diseño que se busca.

Por un lado debe proporcionar flexibilidad a la hora de incorporar distintas soluciones ya existentes (FUF, Mikrokosmos, KPML...) para resolver etapas particulares del procesamiento requerido. El interfaz `Stage` junto con la estructura genérica `Draft` permiten conectar cualquier tipo de módulo al resto del sistema minimizando el conocimiento mutuo que deban tener las partes conectadas. Para poner a prueba esta hipótesis se ha recurrido a jFUF (Chu & Duboe, 2003), una biblioteca de clases Java organizadas en paquetes que proporcionan una API para la definición de los elementos y estructuras típicos de la fase de realización superficial y efectúan llamadas a un servidor Lisp para implementar su comportamiento. Para ello se convierten los distintos objetos Java en estructuras FUF que pueden ser interpretadas por el servidor. Existe un acoplamiento de datos por cuanto que la implementación específica que se utilice en este caso para la clase `Draft`

debe utilizar representaciones semánticas compatibles con la notación FUF.

Por otro lado, se debe permitir engarces distintos y variados de los módulos planteados para poder implementar el espectro completo de tipos de arquitectura descritos. La combinación de patrones utilizada para implementar `Stage` y `ControlFlow` da una respuesta adecuada a este problema.

Asímismo, debe proporcionarse la posibilidad de integrar distintas soluciones en híbridos de textos enlatados, plantillas y generación profunda. La solución basada en `Draft` permite el paso entre los módulos de estructuras de datos parcial o totalmente completadas, facilitando este tipo de integración.

Finalmente deben darse facilidades para que distintos desarrolladores utilicen distintas estructuras fundamentales de datos, acordes, por ejemplo con las teorías lingüísticas de preferencia. La posibilidad de incluir implementaciones propias de la clase `Draft` da respuesta a esta necesidad.

6 Conclusiones y trabajo futuro

A la vista del análisis de las fuentes seleccionadas para obtener una primera especificación de requisitos para un *framework* de estas características, parece evidente que una solución de este tipo presenta múltiples ventajas que reducirían una parte importante de la problemática inherente al desarrollo de aplicaciones de GLN hasta la fecha.

Con objeto de comprobar esta hipótesis, están en marcha ya varios proyectos de implementación de aplicaciones GLN basadas en el código que se ha desarrollado como primer prototipo de este *framework*. Uno de ellos está dedicado a explorar el papel de la generación de lenguaje natural como herramienta para facilitar la navegación por espacios virtuales a usuarios inexpertos. Otro de ellas está centrado en explorar el potencial de la GLN a la hora de mediar entre autores e interactores en medios digitales interactivos. En ambos casos se trabaja sobre problemas reales cuya solución puede contribuir a mejorar el acceso de los usuarios a la enorme masa de información disponible actualmente a través de medios digitales, cuyo volumen está en constante crecimiento.

Bibliografía

Calder, J. et al: *Free choice and templates: how to get both at the same time. In "May I speak freely?"* Between templates and free choice in natural language generation, number D-99-01. Saarbr¨ucken, pp. 19–24.1999.

Cahill, L. et al: *The RAGS Reference Manua*" 2001. Technical Report ITRI-01-07, ITRI, University of Brighton.

Chu, H., Duboe, P.A,: *jFUF, An alpha release of an API to access FUF/SURGE functionality in Java*, *http://www.cs.columbia.edu/~pablo/project/*

DeSmedt, K. et al: Architectures for Natural Language Generation: Problems and Perspectives", in: G.Adorni y M. Zock (eds.) Trends in natural language generation: An artificial intelligence perspective (LNAI 1036) pp. 17-46. Berlin: Springer, 1996

Gamma, E. et al: *Dessign Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995

Hovy, E.: Generating natural language under pragmatic constraints. Hillsdale, NJ: Earlbaum, 1988.

Johnson, R. E., Foote, B. (1988). Designing Reusable Classes. J. of Object-Oriented Programming, vol. 1, 2.

Kantrowitz, M., Bates, J.: Integrated natural language generation systems. In R.Dale, E.Hovy, D. Rösner, and O. Stock, editors, Aspects of Automated Natural Language Generation, pp. 13--28. Springer, Berlin, 1992.

Mann, W. Thompson, S.: Rhetorical Structure Theory: Towards a Functional Theory of Text Organization. In: TEXT, 8(3) (1988)

Reiter, E., Dale, R. *Building Applied Natural Language Generation System*", Cambridge University Press, 2000.

Robin, J.: Revision-Based Generation of Natural Language Summaries Providing Historical Background, PhD Thesis, Columbia University, 1994.