Paradigmas de la Programación
Laboratorio 5: Programación Orientada a Objetos

Laura Alonso Alemany Cristian Cardellino Ezequiel Orbe

En este laboratorio desarrollaremos un proyecto utilizando el paradigma de programación orientada a objetos, para lo cual implementaremos el corrector ortográfico (*spellchecker*) que hicimos en los Laboratorios 2 y 3 en uno de los lenguajes orientados a objetos más conocidos: `Java`. Nuestro objetivo será poner a trabajar los conceptos más importantes de la programación orientada a objetos y poder comprender sus diferencias con respecto a la programación imperativa y la programación funcional.

Características de la presentación

- El trabajo es en grupo: máximo: **dos integrantes**, mínimo: **dos integrantes**.
- Fecha de Entrega: Hasta las 23:59:59 del 28/05/2015.
- Formato de entrega:
  1. Empaquetar el directorio `spellchecker` en un archivo llamado `<dni>-lab-5.tar.gz`, donde `<dni>` es el DNI de alguno de los integrantes del grupo.
  2. Enviar el archivo `<dni>-lab-5.tar.gz` por mail a la dirección:

     `paradigmas@famaf.unc.edu.ar`

     El título del mail debe decir:`lab-5`. En el cuerpo del mail se deben indicar los integrantes (nombre completo y DNI).
  3. Los trabajos enviados fuera de término serán evaluados de acuerdo a la siguiente Política de Entrega Tardía:
     - Hasta 1 día después del deadline: -20% de la nota.
     - Hasta 2 días después del deadline: -40% de la nota.
     - Hasta 3 días después del deadline: -60% de la nota.
     - Hasta 4 días después del deadline: -80% de la nota.
     - Más de 4 días después del deadline: el trabajo se considera NO entregado y se debe recuperar.
  4. Los trabajos enviados que no cumplan con las condiciones de presentación no serán considerados.

Software Requerido

Para realizar este laboratorio necesitaremos el siguiente software:

- Java Platform, Standard Edition (versión 7 en adelante) (Descargar)
- Eclipse (Descargar)

Código Inicial

Para realizar este laboratorio, utilizaremos el código disponible en:

`http://cs.famaf.unc.edu.ar/materias/paradigmas/node/195`

Al extraer el contenido del archivo, tenemos la siguiente estructura de directorios:

```
spellchecker
├── bin
├── src
│   └── SpellChecker.java
├── .classpath
├── .project
└── .settings
```

Una vez extraídos los archivos, se puede importar el proyecto a Eclipse.

1 Especificaciones del Corrector Ortográfico

El corrector ortográfico (spellchecker) que desarrollaremos deberá utilizar un diccionario de palabras conocidas, el cual se cargará en memoria desde un archivo cuando el programa se inicie, y un diccionario de palabras ignoradas, el cual inicialmente estará vacío.

Dado un documento de entrada, el spellchecker copiará a un documento de salida todas las palabras y los signos de puntuación, consultando interactivamente al usuario sobre cada palabra desconocida.

Dada una palabra desconocida, el usuario podrá aceptarla, ignorarla o reemplazarla.

Si el usuario acepta la palabra, la misma se agregará al diccionario, si el usuario ignora la palabra, esta ocurrencia y las subsiguientes se ignorarán. Finalmente, si el usuario decide reemplazar la palabra, el sistema permitirá que el usuario ingrese una nueva palabra que reemplazará a la palabra desconocida en el documento de salida.

Una vez que todo el documento de entrada haya sido procesado, el programa guardará el diccionario en el mismo archivo desde el cual lo cargó al inicio y dejará disponible un nuevo archivo que contenga el documento de salida.

1.1 Funcionalidades deseadas

1. Leer un diccionario de palabras conocidas desde un archivo.
2. Tener en memoria un diccionario de palabras ignoradas.
3. Leer un documento de entrada, palabra por palabra.
4. No proponer acciones para palabras conocidas.
5. Proponer tres acciones posibles para palabras desconocidas: `aceptar`, `ignorar`, `reemplazar`.
6. En caso de aceptar una palabra desconocida: incorporarla al diccionario de palabras conocidas y que las subsiguientes ocurrencias de la palabra sean tratadas como conocidas.
7. En caso de ignorar una palabra desconocida: incorporarla al diccionario de palabras ignoradas e ignorar la palabra y sus subsiguientes ocurrencias.
8. En caso de reemplazar una palabra desconocida: en el documento de salida se escribirá la palabra por la que se reemplaza, en lugar de la original.
9. Actualizar el diccionario de palabras conocidas incluyendo las palabras aceptadas.
10. Escribir un documento de salida con las palabras originales o sus modificaciones, según las acciones realizadas por el usuario.

2 Consignas

La Figura 1 muestra el diagrama de clases de la aplicación. El mismo indica las clases que necesitarán implementar, así como las interacciones entre las mismas. Para cada clase se especifican sus métodos y sus atributos con su correspondiente nivel de visibilidad.

- Implementar las clases `Word` y `WordSet` en el package `word`.
  - La clase `Word` representa una palabra y es un wrapper sobre el tipo de datos `String`. Tiene un atributo privado `word`, y un par setter/getter para el mismo. Sobreescribe los métodos `hashCode` y `equals` de la clase base `Object`. Provee dos constructores, uno de los cuales recibe como argumento un `String` y retorna una instancia que contiene dicho `String`.
  - La clase `WordSet` representa un conjunto de palabras, el cual se implementa internamente con una colección que implemente la interfaz `Set`. El método `iterator` devuelve una referencia a un iterator correspondiente a la implementación interna del conjunto.
- Implementar las clases `Dictionary`, `FileDictionary` y `MemDictionary` en el package `dictionary`.
  - La clase `Dictionary` es una clase abstracta que representa un diccionario de palabras. Las palabras se almacenan como un conjunto de palabras. El método `fromStringList` inserta en el diccionario todas los elementos pertenecientes a la lista pasada como argumento. El método `toStringList`, realiza la operación inversa, es decir, retorna en una lista de strings, todas las palabras que pertenecen al diccionario.
  - La clase `FileDictionary` representa un diccionario que se carga desde un archivo de texto. Es una subclase de `Dictionary`, y además implementa los métodos `load` y `save`, los cuales cargan el diccionario desde un archivo y guardan el diccionario a un archivo respectivamente. El método `save` provee dos implementaciones, una que recibe el `path` del archivo donde se guardará el diccionario,y otra que no recibe dicho argumento y guarda en el diccionario en el mismo archivo desde el cual se cargó. Provee de dos constructores, uno de los cuales recibe un path al archivo, y retorna un diccionario que contiene las palabras en dicho archivo.
  - La clase `MemDictionary` representa un diccionario que se almacena en memoria solamente.
- Implementar la clase `Document` en el package `document`.
  - La clase `Document` representa al documento que se va a procesar. El método `getWord`, lee el documento de entrada, palabra por palabra, copiando al documento de salida todos los caracteres no alfabéticos precedentes que encuentre. Al llegar al final del archivo, lo señaliza mediante una excepción `EOFException`. El método `putWord`, escribe una palabra dada en el documento de salida.
- Implementar la clase `SpellChecker`. La misma no pertenece a ningún package.
  - La clase `Spellchecker` es el punto de entrada de la aplicación. Implementa tres métodos estáticos: `main`, `processDocument` y `consulUser`.

- No se puede modificar la arquitectura de la aplicación, eso implica que no se pueden modificar las interfaces públicas de las clases, ni las dependencias entre las mismas.
- Se pueden utilizar métodos auxiliares, pero no pueden ser públicos.
- Se penalizará la duplicación de código. Si algo se puede heredar, no se implementa de nuevo. Si ya existe un método que realice una tarea, se lo utiliza.

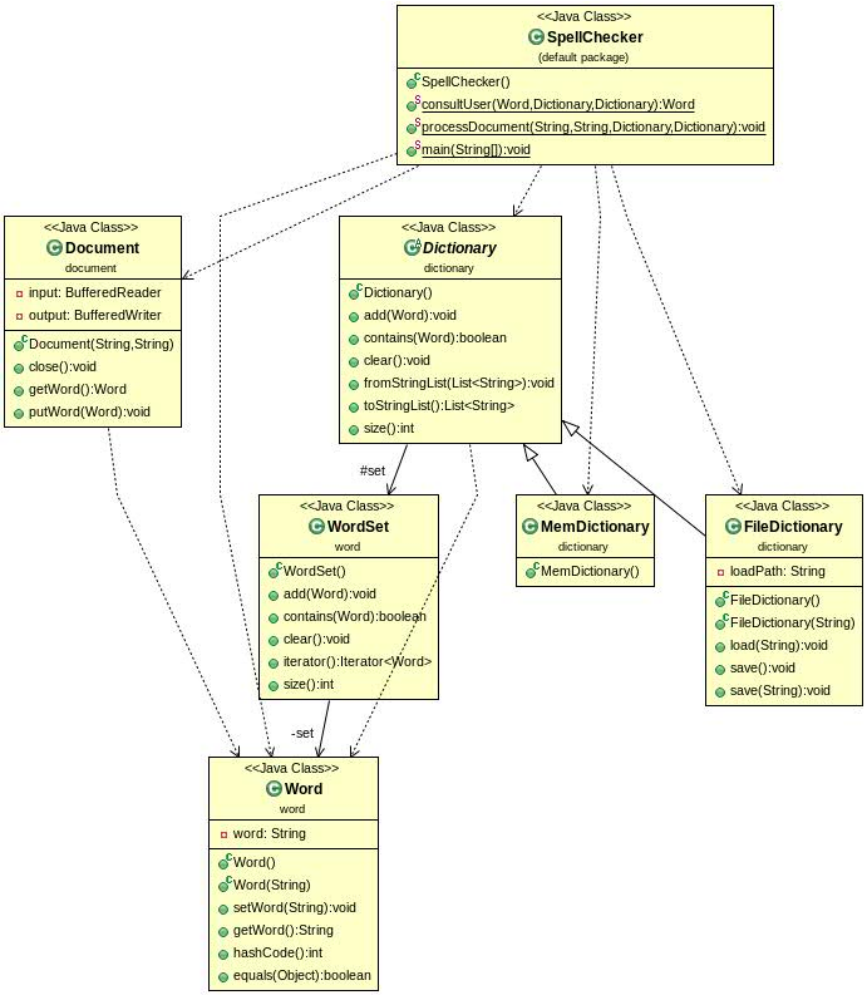

Figure 1: Diagrama de Clases

- Las excepciones deben ser gestionadas.
- Se debe respetar en todo momento el principio de encapsulación, esto es, el acceso a los atributos de cada clase, siempre se realiza a través de setters/getters. Sólo en el caso de una jerarquía de clases, se permite el acceso directo a los atributos, es decir, el acceso de una subclase a los atributos de su superclase.
- Se deben seguir las convenciones de nomenclatura dadas aquí.
- Se deberán documentar las clases utilizando Doc Comments. Estos comentarios deberán servir para generar la documentación de la API en formato HTML mediante el uso de javadoc.[1].
- Es fundamental leer la documentación de Java.

[1]Para más info ver:Javadoc en Wikipedia