Sesión 5

Diseño modular en C++ (I)

5.1. Diseño modular

En esta sesión veremos un primer ejemplo de diseño modular completo. Recordemos que la metodología del diseño modular que usaremos en la asignatura consta de cinco fases:

1. Detectar las clases de datos implicadas en nuestro programa, a partir del enunciado
2. Obtener un esquema preliminar para el programa principal
3. Especificar las clases anteriormente detectadas
4. Escribir el programa principal detallado, usando objetos e instrucciones reales
5. Implementar las clases

Las fases 3 y 5 pueden repetirse durante la implementación, si aparecen nuevas clases auxiliares que no eran evidentes al principio del diseño. También puede ser necesario realizar algunos ajustes en la ubicación o en la especificación de las operaciones, si las decisiones originales no son del todo correctas.

En el ejemplo que vamos a ver, os porporcionamos una parte del diseño para que lo completéis. A continuación, probaremos el programa resultante con juegos de pruebas diseñados para buscar errores en situaciones especiales. También introduciremos una herramienta de documentación llamada *doxygen*.

5.2. Primer caso de estudio: Factor PSI

Dado un texto terminado en '.', obtener la frecuencia de la palabra más frecuente del mismo. El texto consta de una o más palabras formadas por letras o números y separadas por cualquier otro caracter. El texto no contendrá más de `MAXNUMPAL` palabras distintas, cada una de ellas no mayor de `MAXLONG` caracteres. El texto se introduce por el canal standard de entrada. Podéis hacer las

primeras pruebas escribiéndolo por teclado, pero resultará interesante que también lo probéis redireccionando la entrada a un fichero como `pro2_s52.txt`.

De este enunciado podemos deducir dos claras abstracciones de datos: necesitamos la clase `Palabra` para ir leyendo y procesando cada palabra del texto y necesitamos otra estructura para almacenar las diferentes palabras junto con sus frecuencias.

La clase `Palabra` está preparada para usarse en este ejercicio ya que la longitud máxima que permite coincide con el `MAXLONG` de las palabras del texto. Podéis ver un ejemplo de uso en el fichero `ejpalmarca.cc`. Para representar la otra estructura de datos os ofrecemos la clase `ListaPalabras` que también está preparada para usarse en este ejercicio ya que el número de palabras distintas que permite coincide con el `MAXNUMPAL` del texto.

5.2.1. Documentación generada por el programa Doxygen

La especificación de ambas clases se puede encontrar en los formatos generados por doxygen en la subcarpeta `DOC`. Ésta, a su vez, contiene las carpetas `html` y `latex`. Si queréis ver la documentación en `html`, id a la carpeta correspondiente y abrid el fichero `index.html`. Si queréis ver una versión en `pdf`, id a la carpeta `latex` y abrid el fichero `refman.pdf`.

5.2.2. Ejercicio: programa principal

El programa principal tendrá el siguiente esquema:

```
inicializar lista
leer primera palabra
mientras no palabra_vacia hacer
  anadir palabra a lista
  leer palabra
fmientras
obtener frecuencia máxima
```

Implementad el programa principal usando como base el fichero `pro2_s52.cc`, ya que contiene una plantilla que servirá de ejemplo para documentar el programa mediante doxygen. Los ficheros `.hh` y `.o` de las clases `Palabra` y `ListaPalabras` se encuentran en `INCLUSIONS` y `OBJECTES`.

Al ejecutar el programa notaréis que van apareciendo en la pantalla determinados mensajes sobre el progreso del mismo. Dichos mensajes proceden de la implementación que os proporcionamos de la clase `ListaPalabras`. Cuando programéis la vuestra podréis incluirlos mientras realizáis las ejecuciones de prueba, pero en la versión final deberéis eliminarlos.

Probad el programa con los textos que queráis, además de con el fichero `pro2_s52.dat`.

5.2.3. Ejercicio: juegos de pruebas especiales

Producid una serie de ficheros que contengan juegos de pruebas especializados en situaciones límite detectables a partir de la especificación de `ListaPalabras`, por ejemplo

- una lista de tamaño 1
- que la palabra de frecuencia máxima sea la primera del texto
- que la palabra de frecuencia máxima sea la última del texto

5.2.4. Uso del doxygen

En primer lugar, copiad los ficheros de la subcarpeta `HH` a la carpeta de la sesión. Dichos ficheros son `Palabra.hh` y `ListaPalabras.hh`. Son los que se han usado para generar la documentación. Notad que su estado es distinto: `Palabra.hh` ya está completo, con los campos de la representación incluidos, mientras que `ListaPalabras.hh` no contiene dichos campos (se los añadiremos en el ejercicio 5.2.5) y no sirve para compilar el programa principal. Notad, por lo tanto, que para usar doxygen no es necesario que los programas y las clases sean definitivos.

A continuación vamos a usar el programa doxygen para generar una nueva version de la documentación del ejercicio, de forma que incluya el programa principal completo. Para ello, aplicaremos los pasos descritos en el fichero `docu_doxygen.txt`, que tendréis que haber descargado junto con el `pdf` de la sesión.

En esencia, documentar una clase o un programa mediante doxygen solo requiere introducir en éstos unos comentarios especiales, acotados por los caracteres `/**` y `*/`. Dichos comentarios serán procesados por doxygen para producir la documentación en los formatos que habéis visto en las carpetas `html` y `latex`.

Las opciones de configuración de doxygen se definen en un fichero habitualmente llamado `Doxyfile`. El que os incluimos en la carpeta de la sesión contiene todas las opciones necesarias para documentar las especificaciones del laboratorio y de la práctica de la asignatura. En esta sesión solo debéis tocarlo para modificar los datos identificativos de vuestro programa, definir las rutas a vuestras carpetas de trabajo o el idioma de vuestros documentos.

Observad que la documentación resultante incluye un diagrama modular de toda la solución y otro por cada clase. Dichos diagramas se construyen a partir de los `#include` de los ficheros implicados. Si queremos añadir o eliminar elementos tendremos que usar algún `#include` redundante o anular temporalmente alguno de los `#include` necesarios. Por ejemplo, aseguraos de que en los diagramas de la versión que acabáis de obtener no aparecen `vector` ni `utils.PRO2`.

5.2.5. Implementación de `ListaPalabras`

Completad la parte privada del fichero `ListaPalabras.hh` con una elección adecuada de campos. Una posibilidad es

```
private:
  struct palfrec {
    Palabra par;
    int freq;
  };
```

```
static const int MAXNUMPAL = 20;
vector<palfrec> paraules; // vector que guarda cada palabra y su frecuencia
int nparaules; // las posiciones relevantes son paraules[0 .. nparaules-1]

/* Invariante de la representación:
   - 0 <= nparaules <= longitud_maxima () = MAXNUMPAL
   - los campos frec de paraules[0 .. nparaules-1] son > 0
   - los campos par de paraules[0 .. nparaules-1] no tienen palabras repetidas
*/
```

Producid vuestro propio fichero `ListaPalabras.cc` y comprobad que el principal sigue funcionando si lo linkamos con el `ListaPalabras.o` correspondiente. Usad los mismos juegos de pruebas que en los ejercicios anteriores.

5.2.6. Ejercicio: otros juegos de pruebas

Producid otra serie de juegos de pruebas especializados en situaciones límite relacionadas con elementos específicos de la implementación de `ListaPalabras`. Comprobad que vuestro programa los pasa correctamente.

Algunas situaciones interesantes son:

- que la palabra de frecuencia máxima esté en la última posición ocupada de la lista
- llenar una lista (introducir `MAXNUMPAL` palabras distintas: usad un `MAXNUMPAL` pequeño) y, a continuación, añadir una palabra que no esté en la posición `MAXNUMPAL`
- añadir a una lista llena una palabra que esté en la posición `MAXNUMPAL`

5.3. Ejercicio: un cambio en la especificación

Reprogramad la operación `anadir_palabra` para que su precondición no requiera saber si el nuevo elemento ya existe en la lista o si ésta ya está llena. Su especificación queda así

```
void anadir_palabra (const Palabra & p, bool &b);
 /* Pre: cierto */
 /* Post: b indica si p aparece en el parámetro implícito original;
    si b o el p.i. original no estaba lleno, p se ha añadido al p.i.
    (si b, su frecuencia se incrementa en 1; si no, su frecuencia es 1);
    en otro caso, el p.i. no cambia */
```

Comprobad el buen funcionamiento de la nueva versión con datos adecuados.

5.4. Ejercicio: optimización de `max_frec`

Supongamos que la operación `max_frec` ha de tener prioridad absoluta en cuanto a eficiencia. Para optimizarla al máximo, podemos introducir un campo nuevo a la clase `ListaPalabras` que contenga en todo momento la posición de la palabra más frecuente, de forma que devolver su frecuencia sea inmediato. Sin embargo, esto obliga a modificar algunas de las demás operaciones, especialmente `anadir_palabra`, ya que cada palabra que se añada puede convertirse en la más frecuente, en lugar de la que se había obtenido previamente.

Aplicad todos los cambios necesarios para conseguir esta optimización (sin perjudicar las demás operaciones más de lo necesario) y actualizad el invariante de la representación. Repetid todas las pruebas anteriores y realizad una nueva ronda de pruebas, específica para los cambios que hayáis introducido en las operaciones.

Realizad una segunda versión del programa con un `main` en forma de menú, que permita aplicar esta operación cualquier número de veces en una misma ejecución:

- añadir una palabra a la lista de palabras (opción -1)
- consultar la frecuencia de la palabra más frecuente de la lista (opción -2)

La entrada del programa ya no será un texto marcado, sino una secuencia de la forma

```
-1 casa  // introducir la palabra ``casa'' a la lista

-1 hola  // introducir la palabra ``hola'' a la lista

-2 // obtener la frecuencia de la palabra más frecuente de la lista

....

-3  // final de programa
```

Tenéis un ejemplo de datos y resultados en el fichero `pro2_s54.dat`. Probad el programa con otros datos, explorando situaciones de empate, lista llena, etc.

5.5. Ejercicio: las *n* palabras más repetidas

Dado un texto terminado en '.' y un número *n* mayor que 0, obtener la lista de las *n* palabras más frecuentes del mismo, ordenadas de mayor a menor frecuencia. Si el número de palabras distintas del texto es menor o igual que *n*, se obtendrán todas. Si hay varias palabras con la misma frecuencia, tendrán prioridad las palabras que alcanzaron primero dicha frecuencia.

Ejemplo: si *n* es 3 y el texto es ``q w e r r e w q w e q r''., el resultado deberá ser:

```
w 3
e 3
q 3
```

Suponed que esta operación se utilizará con bastante frecuencia, por lo que no resultará rentable ordenar la estructura cada vez. En vez de eso, modificad la operación `añadir` para que mantenga ordenada la estructura y actualizad el invariante de la representación.

Notad que este programa requerirá, entre otras cosas, modificar la cabecera de la operación `escribir_lista`, para poder pasarle la *n* como parámetro. Ello os obligará a editar vuestro fichero `ListaPalabras.hh`.

Probad todos los casos especiales que podáis identificar: que la nueva palabra no esté en la lista, que si lo esté y avance posiciones, que no avance, etc.

Realizad una segunda versión del programa con un `main` en forma de menú que permita añadir palabras a la lista y obtener las n palabras más frecuentes cualquier cantidad de veces.

- añadir una palabra a la lista de palabras (opción -1)
- consultar las n palabras más frecuentes de la lista (opción -2)

La entrada del programa ya no será un texto marcado, sino una secuencia de la forma

```
-1 casa  // introducir la palabra ``casa'' a la lista

-1 hola  // introducir la palabra ``hola'' a la lista

-2 1 // obtener la palabra más frecuente de la lista

-1 mesa

-1 silla

-2 2 // obtener las 2 palabras más frecuentes de la lista

....

-3  // final de programa
```

Tenéis un ejemplo de datos y resultados en el fichero `pro2_s55.dat`. Probad el programa con otros datos, explorando situaciones de empate, lista llena, recolocaciones extremas de las palabras, etc.