Sesión 9

Diseño modular en C++ (III)

En esta sesión presentamos una práctica de diseño modular más compleja que las anteriores. Empleamos la misma metodología de diseño introducida en la sesiones 5 y 8 y también, como entonces, especificamos las clases usando el doxygen. Sin embargo, esta vez también incluiremos la implementación de la práctica en el documento generado.

Previamente, mostraremos los cambios en el fichero `Doxyfile` necesarios para lograr dicho documento y los probaremos sobre las soluciones de las sesiones 5 y 8.

Por último, plantearemos algunas modificaciones de la solución original de la práctica de la sesión, como ejercicio.

9.1. Documentación completa mediante doxygen

Nuestro objetivo final es producir documentos que contengan toda la solución de nuestras prácticas: especificación e implementación. Para ello, solo hemos de modificar dos elementos del fichero `Doxyfile` ya conocido. En primer lugar, hay que decirle a doxygen que incorpore los elementos privados de las clases. En segundo lugar, que procese *todos* los ficheros `.cpp`, no solo el del `main` como hasta ahora. En el `Doxyfile` de esta sesión ya se han aplicado dichos cambios.

Por lo demás, mantendremos la política de solo incluir comentarios doxygen sobre las operaciones en los ficheros `.hpp` de las clases. En particular, deberemos añadir las especificaciones de las operaciones privadas y los comentarios que consideremos oportunos sobre los campos de cada clase. En los ficheros `.cpp` de las clases solo usaremos la etiqueta `@file`.

9.2. Ejercicio: documentación completa de los ejemplos anteriores

Copiad el fichero `Doxyfile` de esta sesión a las carpetas de las sesiones 5 y 8 y adaptadlo para producir la documentación completa de los ejercicios de las mismas. Partid de los ficheros definitivos y añadid las especificaciones pre/post de las operaciones privadas a los ficheros `.hpp` y la etiquetas `@file` a los `.cpp`.

9.3. Práctica: Experimentos inmunológicos

En el fichero `inmuno.pdf` encontraréis el enunciado del ejercicio (también está publicado en la web de PRO2 como ejemplo de práctica resuelta). Esencialmente se trata de llevar a cabo una serie de experimentos con organismos celulares para estudiar sus propiedades inmunológicas.

En la carpeta `SOL_ORIG` están los ficheros `hpp` y `cpp` de las clases y el `pro2.cpp` de una posible solución. El fichero `Doxyfile` está modificado para que el documento obtenido sea completo.

En la carpeta `joc_de_proves_orig` está el documento explicativo sobre el formato de los juegos de pruebas de la práctica y la entrada y la salida del juego de pruebas público.

Podéis inspiraros en todo este material de cara a la práctica que estáis realizando como parte de la evaluación de PRO2. A continuación, repasaremos algunas de las situaciones que aparecen y propondremos algunos ejercicios.

9.4. Paso de la especificación a la implementación

En primer lugar, notad como el programa principal ya está adaptado al formato de los datos y resultados mostrado en el documento correspondiente.

Los ficheros `hpp` de las clases ya contienen los elementos privados de cada una de ellas: observad los campos (y sus comentarios) y las operaciones privadas (y sus especificaciones). Éstas han ido surgiendo al implementar cada clase en su correspondiente fichero `cpp`.

Cuando implementamos una operación pública de cierta entidad, normalmente nos interesa descomponerla en operaciones auxiliares. A veces éstas también se usarán en otras clases pero, si ese no es el caso, dichas operaciones no han de aparecer en la especificación de la clase correspondiente como públicas, sino como privadas (y `static`, si es necesario).

Considerad por ejemplo la operación `anadir_organismo` de la clase `Sistema`. Simplemente, se encarga de organizar sus tareas de más nivel, dejando que el trabajo más pesado lo realicen sus auxiliares. En particular, dado que un sistema contiene dos colas de organismos, y cada vez que un organismo entra en el sistema hay que probarlo contra todos los de una de ellas, el uso de la auxiliar `luchas_org_cola` nos evita programar ambos recorridos, ya que ella se encarga de realizar el recorrido necesario en cada momento, pues está convenientemente parametrizada. La misma política se aplica con la operación `escribir` y su auxiliar `escribir_sistema_cola`.

Un caso particular especialmente interesante de esta estrategia de descomposición se presenta en la implementación de la clase `Organismo`. Las células de un organismo se estructuran de manera arbórea, razón por la cual uno de los campos de la clase es un árbol. Comprobad como, cuando llega el momento de tratar los árboles, las operaciones correspondientes han sido definidas como auxiliares (y por lo tanto privadas). Eso no solo conlleva las ventajas expuestas para el caso anterior sino que, además, facilita el diseño recursivo de dichas operaciones pues, al manejar solamente los árboles, podemos plantear las llamadas recursivas directamente con sus hijos izquierdo y derecho, etc. Notad que esto es lo que hacemos con la operación `lucha_organismos` y sus auxiliares `simetricos` y `lucha_arboles`, y también con las operaciones de lectura y escritura de organismos (observad que en este caso es obligatorio copiar el árbol antes de llamar a

la auxiliar que lo trata).

Por otra parte, veréis que aunque son las células las que tienen tamaño `N`, las operaciones de lectura de `Sistema` y `Organismo` también tienen como parámetro dicho valor. Así podemos ir "transportándolo" hasta el momento de crear cada célula nueva en cada lectura. Otra opción es dimensionar el vector de cada célula en la misma creadora, no en el momento de la lectura, pero entonces habría que definir una versión de dicha creadora con parámetro `N`.

Por último, notad también que, como las células se guardan en los nodos del árbol de cada organismo, hay que definir el concepto de "célula vacía", para hacer de marca de final de lectura para los árboles (dicho de otro modo, para reconocer cuándo leemos un árbol de células vacío). La decisión de guardar una célula en cada nodo viene dada porque los identificadores de las células no facilitan otras opciones más eficientes, como por ejemplo, guardar solo el identificador en el nodo y añadir un vector de células separado del árbol.

9.5. Ejercicios

Después de familiarizaros con la estructura de la solución de la práctica, sus componentes más importantes y sus juegos de pruebas, aplicad las modificaciones que describimos a continuación.

- Emplead dos listas en lugar de dos colas para implementar `Sistema`, de forma que en la operación `luchas_org_cola` se pueda evitar el uso de la operación privada `recolocar`. La clave consiste en usar correctamente la operación `erase` de las listas.
- Suponed que los organismos del sistema tienen un número máximo de `M` (un entero mayor que 0) células y que los identificadores de las mismas pertenecen al intervalo `[1..M]` (recordad que dentro de un organismo los identificadores de células no se repiten y que dos células de distintos organismos con el mismo identificador no tienen por qué ser iguales). Reimplementad `Organismo` de forma que se aproveche esta propiedad: en lugar de un árbol de células, emplead un árbol de enteros y un vector de células de tamaño `M`. De esta forma, algunas operaciones privadas de organismos no necesitan usar el vector y las copias de árboles son más eficientes. Además, en este caso, una célula no ha de guardar su identificador.
- Modificad la política de luchas entre organismos, de forma que se enfrenten los organismos que tengan estructuras celulares iguales en lugar de estructuras celulares simétricas. Al enfrentarse dos organismos, cada célula de uno se enfrentará a la que ocupa la misma posición en el otro.

Disponéis de un ejemplo de datos de entrada y su salida correspondiente en la carpeta `joc_de_proves_exercici`. Notad que el valor `M` de cada ejecución ha de proporcionarse adicionalmente como dato de la misma.