Práctica 2
Google Earth

Programación Orientada a Objetos
Curso 2010-2011

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Antonio M. Corbí

1. Introducción

En esta segunda práctica seguiremos trabajando con los archivos KML de la primera práctica; para ello vamos a extender la información que podemos almacenar en ellos. En este sentido haremos uso de la característica de herencia que tienen los lenguajes orientados a objetos.

Además se pretende que el alumno aprenda a usar otras características del lenguaje C++ que le serán útiles a la hora de escribir programas con este lenguaje.

1.1. Objetivos

Los objetivos que se pretenden conseguir con esta segunda práctica son los de:

- Hacer uso de *herencia*.
- Emplear la característica de *enlace dinámico*.
- Hacer uso de *clases abstractas*.
- Gestionar los errores en tiempo de ejecución mediante el uso de *excepciones*.

1.2. Planteamiento

Consultando la documentación de KML nos hemos dado cuenta que la clase `Posicion` que hemos creado en la práctica anterior es en realidad una subclase de otra clase llamada `Recurso`, más general.

`Posicion` hereda de la clase `Recurso` las variables de instancia *name* y *description*. `Posicion` añade a estos datos lo relativo a las *coordenadas* de la posición.

Derivando también de `Recurso` nos encontramos con la clase `Imagen`, la cual nos permite fijar imágenes a una posición en la pantalla, independiente del mapa que se esté visualizando.

Este hecho tiene una serie de consecuencias en nuestro diseño:[1]

[1] *Nota*: Cambiar un nombre de atributo, método,... no es trivial. Se debe hacer en cualquier punto del código que use dicho elemento. Esto forma parte de un conjunto de técnicas que realizan cambios en el código llamado *refactorización*. Si usas un entorno de desarrollo moderno, como *Eclipse* (http://www.eclipse.org), que puedes encontrar instalado en las máquinas de los laboratorios de la EPS, puedes hacerlo de forma automática y sin correr riesgos.

- El método "`void Posicion::leerPosicion (std::ostream&)`" ahora pasa a ser "`virtual void Recurso::leer (std::ostream&) = 0;`"[2] y las clases derivadas de `Recurso` le proporcionan una implementación apropiada.
- Lo mismo pasa con el método "`void Posicion::leerPosicion (const std::string&)`" ahora pasa a ser "`virtual void Recurso::leer (const std::string&) = 0;`"[3], por tanto las clases derivadas de `Recurso` le proporcionan una implementación apropiada.
- Algo similar ocurre con "`void mostrarKML (std::ostream&)`" y con "`void mostrar (std::ostream&)`" aunque estos mantienen el mismo nombre.
- La clase `Recurso` añade un nuevo método:
  "`virtual bool admiteDistancia () = 0;`", el cual sirve para saber si un objeto instancia de una clase derivada de `Recurso` sirve para calcular distancias a él o desde él
- La `Aplicacion` ya no almacenará `Posicion`es sino `Recursos`[4]... por tanto el método "`Posicion Aplicacion::getPosicion (int)`" ahora pasa a ser "`Recurso* getRecurso (int)`"; por tanto ya no *busca posiciones* sino que *busca recursos*: "`void Aplicacion::buscarRecurso (const std::string&, std::ostream&);`"

Los datos específicos de una `Imagen`, representada en formato KML son los siguientes:[5]

```
<ScreenOverlay>
  ...
  <Icon> <href>...</href></Icon>
  <overlayXY x="double" y="double"/>
  <screenXY x="double" y="double"/>
  <rotationXY x="double" y="double"/>
  <size x="double" y="double"/>
  <rotation>double</rotation>
</ScreenOverlay>
```

Para comprender que función tiene cada etiqueta, (lo cual no es imprescindible para realizar la práctica), puedes consultar la documentación de KML[6].

Como se indica en ella, la imagen a superponer viene especificada por la etiqueta <Icon>, p.ej.:

```
  <Icon>
    <href>http://miservidor/miimagen.jpg</href>
  </Icon>
```

[2] Date cuenta del cambio de nombre: `leerPosicion` → `leer` y de que es `virtual puro`.
[3] Date cuenta también del cambio de nombre
[4] Realmente guardará punteros a `Recursos`.
[5] `double` indica que el valor admitido es un número real.
[6] http://code.google.com/intl/es-ES/apis/kml/documentation/kmlreference.html#screenoverlay

De lo aquí dicho se deduce que la clase `Recurso` será una clase abstracta de la que derivan `Imagen` y `Posicion`.

Por tanto deberemos refactorizar el código escrito en la práctica 1 para la clase `Posición` en la nueva clase `Recurso` y añadir la nueva clase `Imagen`. Además, allí donde tenga sentido aplicaremos la característica de enlace dinámico que nos proporciona `C++`.

1.3. Excepciones

Con el fin de obtener un código más robusto reescribiremos parte del código destinado a la gestión y tratamiento de errores mediante el uso de *excepciones*.

Para trabajar con excepciones nos basaremos en la jerarquía de clases de excepciones estándar que nos proporciona el lenguaje. Incluiremos la cabecera <`stdexcept`>, la cual nos proporciona diversas clases para representar excepciones, entre ellas:

`exception`**:** Representa la clase base de la jerarquía de clases de excepciones estándar de C++.

`runtime_error`**:** Representan errores producidos en tiempo de ejecución y que se salen del ámbito del programa.

En esta práctica crearemos nuestras propias clases de excepciones. *Obligatoriamente*, derivaremos éstas de la clase `runtime_error`. Todas las clases de excepciones estándar admiten un constructor a partir de un parámetro de tipo `std::string` que describe el error producido.

Esta cadena que describe la excepción se puede consultar mediante el método `const char* what()`, pero ten en cuenta que no la devuelve como un dato de tipo `std::string` sino como una cadena de '`C`' acabada en el carácter '`\0`' -como puedes observar por el tipo del resultado-.

Nuestras excepciones tendrán también un constructor con un parámetro de clase "`string`", el cual describirá el error producido (por ejemplo, "`SE3- NO EXISTEN RECURSOS`"). Esta cadena será el mensaje que devolverá el método `what()` al ser invocado. No es necesario implementar la forma canónica completa. Las excepciones que podrá emitir nuestra aplicación se detallan a continuación:

ErrorFicheroNoEncontrado Se lanzará cuando no se pueda abrir un archivo. Emitirá el mensaje "`SE6- EL FICHERO X NO SE HA PODIDO ABRIR`", donde 'X' es el nombre del fichero.

ErrorNoExistenRecursos Se lanzará cuando no haya `Recursos` que mostrar. Emitirá el mensaje "`SE3- NO EXISTEN RECURSOS`"[7].

[7] Este mensaje sustituye al mensaje SE4 de la práctica 1.

ErrorRecursoIncorrecto Se lanzará cuando se trate de calcular la distancia a un `Recurso` o entre dos tipos de `Recursos` que no admiten esta operación. Emitirá el mensaje "`SE4- RECURSO INCORRECTO`".

ErrorRecursoNoEncontrado Se lanzará cuando se esté buscando un `Recurso` por medio de una subcadena y no se encuentre. Emitirá el mensaje "`SE5- RECURSO NO ENCONTRADO`".

En el cuadro **??** puedes ver los métodos que deben lanzar las excepciones indicadas en caso de producirse el error correspondiente.

Excepción	Métodos
ErrorFicheroNoEncontrado	Aplicacion::pedirNombreFicheroRecursos Aplicacion::setEntrada Aplicacion::setSalida
ErrorNoExistenRecursos	Aplicacion::mostrarRecursos Aplicacion::mostratRecursoKML Aplicacion::calcularDistancia Aplicacion::buscarRecurso
ErrorRecursoIncorrecto	Aplicacion::calcularDistancia
ErrorRecursoNoEncontrado	Aplicacion::buscarRecurso

Cuadro 1: Excepciones y los métodos que las lanzan.

Captura de excepciones El código que llama a alguno de los métodos del cuadro **??** será el encargado de capturar las posibles excepciones que estos lancen y mostrar el mensaje asociado por la salida estándar y por el fichero de salida.

1.4. Requerimientos

Al iniciarse el programa, debe mostrarse un menú para que el usuario pueda elegir una opción. Este menú será similar al de la práctica 1, al cual se le añadirá la opción de `Insertar Imagen`, quedando así:

```
1 INSERTAR POSICION
2 INSERTAR IMAGEN
3 MOSTRAR RECURSOS
4 MOSTRAR RECURSO EN FORMATO KML
5 CALCULAR DISTANCIA
6 BUSCAR RECURSO
7 INSERTAR RECURSOS DESDE FICHERO DE TEXTO
99 SALIR
OPCION:
```

Nuestra clase `Aplicación` almacenará como recursos las *posiciones* y las *imágenes* que el usuario añada. Ten en cuenta que determinadas opciones, p.ej. 5, no tienen sentido con *Imágenes*.

Las distintas acciones *nuevas* que tendrá que llevar a cabo la práctica se detallan a continuación.

1.5. Insertar imagen

Cuando se seleccione esta opción, se añadirá una nueva imagen. Para ello, deben pedirse los siguientes datos al usuario, uno a uno:[8]

```
M21-  NOMBRE:
M22-  DESCRIPCION:
M23-  OVERLAY-X:
M24-  OVERLAY-Y:
M25-  SCREEN-X:
M26-  SCREEN-Y:
M27-  ROTATION-X:
M28-  ROTATION-Y:
M29-  SIZE-X:
M210- SIZE-Y:
M211- ROTATION:
M212- URL IMAGEN:
```

1.6. Mostrar recursos

Esta opción muestra por pantalla y en le fichero de salida todas las marcas de posición o imágenes. Si no hay marcas de posición o imágenes se debe lanzar la excepción `ErrorNoExistenRecursos` y volver al menú principal.

En caso contrario, el programa debe mostrar un listado con el formato del siguiente ejemplo:

```
Ballena|Una ballena cerca de la costa|32.66146|-117.23385
Imagen1|Descripción de la imagen1|file://brujula.jpg|1.25|3.43|6.82|4.55|2.00|23.12|1.0|1.0|45.0
Logo Firefox|Logo de Firefox dibujado en el campo|45.123769|-123.113785
```

En este ejemplo, los datos de la imagen de ejemplo "`Imagen1`" se corresponderían con las siguientes etiquetas:

```
nombre|descripcion|icon|overlay-x|overlay-y|screen-x|screen-y|rotation-x|rotation-y|size-x|size-y|rotation
```

[8] No es necesario validar estos datos. Supón que siempre son correctos.

1.7. Mostrar recurso en formato kml

Cuando el usuario elija esta opción, se mostrará por pantalla y en el fichero de salida el listado con todas las posiciones y/o imágenes de la misma manera que ocurre con la opción anterior. Si el listado está vacío se debe lanzar la excepción `ErrorNoExistenRecursos` y volver al menú principal. En caso contrario, se pedirá al usuario el número de la marca de posición y/o imagen mediante el siguiente mensaje:

```
M5- SELECCIONAR RECURSO:
```

Si el número seleccionado[9] no es correcto, se debe emitir el mensaje de error `SE2- DATO X INCORRECTO` y pedirlo de nuevo mostrando el mensaje `M5`.

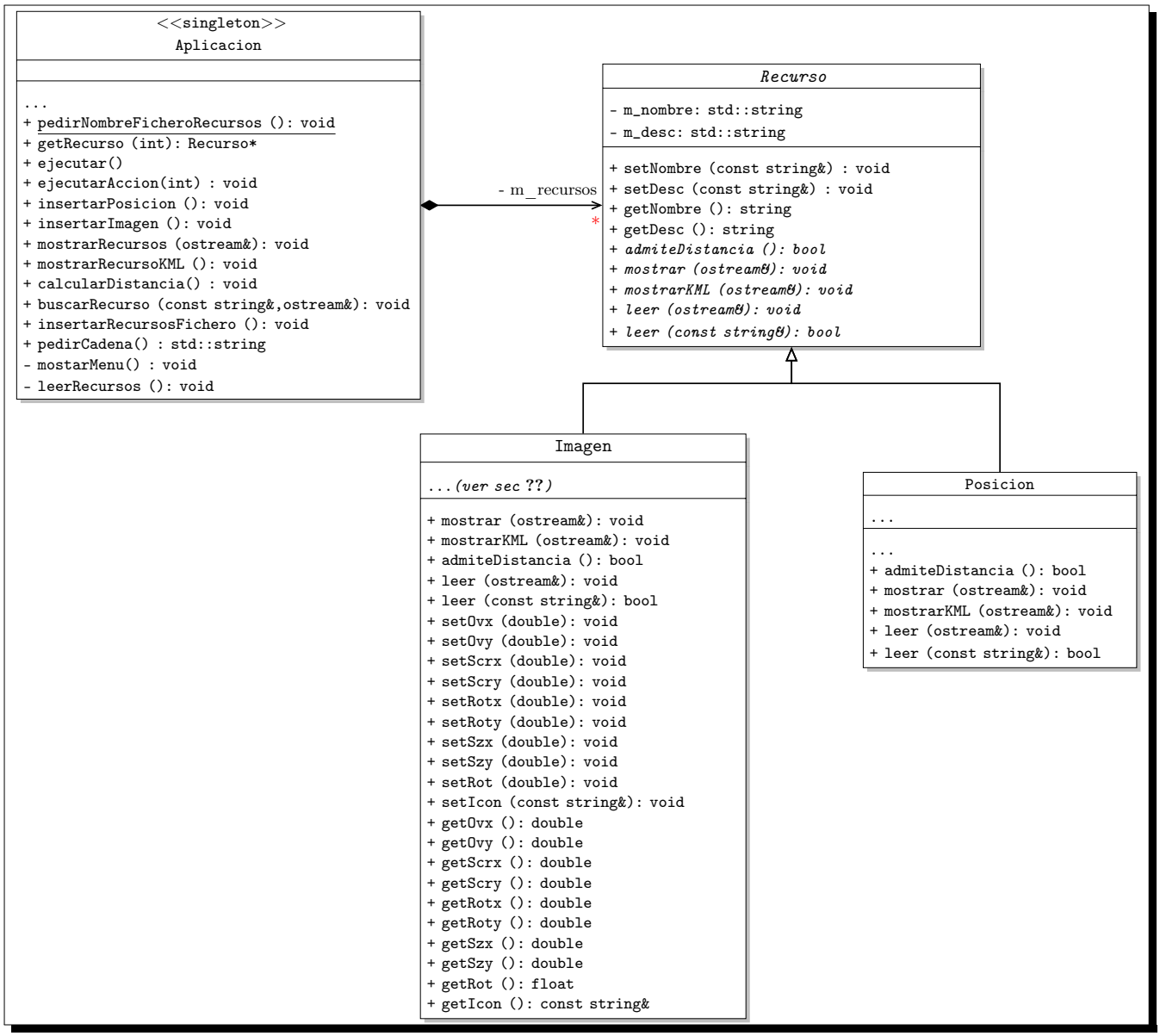
Finalmente, se mostrará, en formato KML, por pantalla y en el fichero de salida la marca de posición o la imagen seleccionada. En el caso de ser una imagen, nos guiaremos por el siguiente ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
   <ScreenOverlay>
      <name>Imagen1</name>
      <description>Descripcion de la imagen1</description>
      <Icon>file://brujula.jpg</Icon>
      <overlayXY x="1.25" y="3.43"/>
      <screenXY x="6.82" y="4.55"/>
      <rotationXY x="2.00" y="23.12"/>
      <sizeXY x="1.0" y="1.0"/>
      <rotation>45.0</rotation>
   </ScreenOverlay>
</kml>
```

2. El diagrama de clases

En este diagrama de clases se muestran *solo las nuevas clases que aparecen y los cambios que pueda haber en el interfaz de clases ya vistas en el diagrama de la práctica 1*[10]. Se han omitido las excepciones.

[9] Se supone que el usuario siempre introducirá un número entero, comenzando por el 0.
[10] Para *Aplicacion* se incluyen todos los métodos de instancia.



3. Interfaz de las clases

En este apartado es aplicable todo lo dicho en el mismo apartado de la práctica 1. Vamos a comentar solamente las diferencias o añadidos que aparecen en esta segunda práctica.

3.1. Aplicacion

void pedirNombreFicheroRecursos () Solicita el nombre de un archivo de recursos. Si el fichero no se puede abrir, lanzará la excepción `ErrorFicheroNoEncontrado`.

void setEntrada () Informa a la aplicación de cuál es el archivo de recursos y los lee. Si no puede abrir el archivo, lanza la excepción `ErrorFicheroNoEncontrado`.

void setSalida () Informa a la aplicación de cuál es el archivo de salida y lo abre en modo escritura. Si no puede abrirlo, lanza la excepción `ErrorFicheroNoEncontrado`.

Recurso* getRecurso (int) Devuelve el puntero al `Recurso` *i-ésimo* de entre todos los que tiene en ese instante la aplicación.

void insertarPosicion () Lee una `Posicion` por la entrada estándar y la guarda como un recurso.

void insertarImagen () Lee una `Imagen` por la entrada estándar tal y como se indica en la sección **??** y la guarda como un recurso.

void mostrarRecursos (ostream& os) Mostrará por la salida estándar y en el archivo de salida `os` los recursos almacenados en ese momento tal y como se indica en la sección **??**.

void mostrarRecursoKML () Mostrará por la salida estándar y en el archivo de salida de la aplicación el recurso seleccionado por el usuario en formato KML. Consulta la sección **??**.

void insertarRecursosFichero () Solicita el nombre de un fichero de recursos y lleva a cabo la lectura de los mismos.

void buscarRecurso (const string& s, ostream& os) Lleva a cabo la búsqueda de una cadena 's' entre los *nombres* y *descripciones* de los Recursos. La salida que produzca la deposita en el stream `os`. Si no encuentra la cadena buscada en ningún recurso lanzará la excepción `ErrorRecursoNoEncontrado`.

void leerRecursos () Una vez abierto el archivo de recursos lleva a cabo la lectura de los mismos, teniendo en cuenta si el recurso es una Posicion o una Imagen y creando el objeto apropiado para guardar en memoria. El formato del archivo de recursos es el mismo que el de la lista de recursos presentada en la sección **??**.

void calcularDistancia () Solicita al usuario que seleccione dos recursos, de forma similar a como se hacía en la práctica 1. Comprobará si los recursos solicitados por el usuario son validos para calcular la distancia entre ellos. En ese caso, calcula la distancia y la muestra por la salida estándar y por el archivo de salida de la aplicación, como se especifica en la práctica 1. Si no se puede calcular la distancia, lanzará una excepción de tipo `ErrorRecursoIncorrecto`.

3.2. Recurso

Todos estos métodos son *abstractos*:

bool admiteDistancia () Indica si un recurso es capaz de calcular su distancia a otro (una `Posicion` lo es, una `Imagen` no).

void mostrar (std::ostream& os) Muestra el recurso en el flujo de salida `os` en el formato especificado en el enunciado.

void mostrarKML (std::ostream& os) Muestra el recurso en el flujo de salida `os` en formato KML tal y como se especifica en el enunciado.

void leer (std::ostream&) Solicita al usuario los datos de un recurso por la entrada estándar. En caso de error en los datos se comporta como se especifica en la sección 1.3 de la práctica 1.

bool leer (const string&) Lee un recurso desde una cadena donde los componentes del recurso están separados por el carácter '|', como en el ejemplo de la sección **??**. Devuelve cierto si los datos son correctos. Si no lo son, el recurso no se modifica y el método devuelve falso.

3.3. Imagen

Los atributos de un objeto `Imagen` se corresponden con cada una de las etiquetas KML de un `ScreenOverlay`: una cadena de caracteres para la etiqueta `Icon` y un atributo de tipo `double` para cada uno de los atributos numéricos del resto de etiquetas (ver sec. **??**).

Esta clase implementa los métodos abstractos heredados de `Recurso` y además define *setters* y *getters* propios. Los puedes ver en el diagrama de clases de la página **??**.

3.4. Posicion

Implementa los métodos abstractos heredados de `Recurso` y además define *setters* y *getters* propios (los relativos al nombre y descripción de la posición pasan a `Recurso`).

Recuerda que algunos métodos de la clase `Posicion`[11] han cambiado de nombre y/o signatura de la práctica 1 a la práctica 2.

[11] Básicamente los heredados de `Recurso`.

4. El programa principal

Sin cambios, excepto que debe capturar las excepciones que se produzcan al tratar de abrir alguno de los ficheros de entrada o salida. Los mensajes de error asociados a estas excepciones sólo es necesario emitirlos por la salida estándar.

4.1. Documentación

Debéis incluir en los ficheros fuente todos los comentarios necesarios en formato Doxygen. Estos comentarios deben estar en su versión corta y detallada, y deben definirse para:

Ficheros debe incluir nombre y dni de los autores.

Clases propósito de la clase: 3 líneas

Operaciones 1 línea para funciones triviales, y 2 líneas + parámetros entrada, parámetros de salida y funciones dependientes para operaciones más complejas.

Atributos propósito de cada uno de ellos: 1 línea

4.2. Estructura de directorios

La práctica debe ir organizada en tres directorios:

include contiene los ficheros de cabecera `Aplicacion.h`, `Menu.h`, `ItemDeMenu.h`, `Recurso.h`, `Imagen.h`, `Excepciones.h`[12], `Posicion.h` y `Punto.h`.

lib contiene los ficheros de código fuente `Aplicacion.cc`, `Menu.cc`, `ItemDeMenu.cc`, `Recurso.cc`, `Imagen.cc`, `Excepciones.cc`, `Posicion.cc` y `Punto.cc`.

src contiene el fichero del programa principal `main.cc`.

Además, al ejecutar la herramienta Doxygen se generará un cuarto directorio '`doc`' que contendrá la documentación en html.

Toda esta estructura de directorios debe estar comprimida en un fichero llamado `poop2-10-11.tgz` que no supere los 300 KB:
`tar czvf poop2-10-11.tgz *`

En la entrega debéis incluir todos los directorios y ficheros (`.h` y `.cc`) menos los ficheros `makefile`, `doxyfile` y la carpeta `doc`.

[12] Un único fichero para todas las excepciones.

5. Normas de entrega, requisitos y aclaraciones

Igual que en la práctica 1.

6. Evaluación

La corrección de la práctica es automática. Esto significa que se deben respetar estrictamente los formatos de entrada y salida especificados en este enunciado, así como la interfaz pública de las clases, tanto en la signatura[13] de los métodos como en el funcionamiento de éstos. Así, por ejemplo, el método `Posicion::mostrarKML(ostream& salida)` imprime el contenido de la posicion con el formato especificado en el flujo '`salida`'.

La nota de esta práctica corresponde al 50 % de la nota final de prácticas.

Además de la corrección automática, se va a utilizar un programa detector de copias. Si se descubre que algún alumno ha copiado una práctica de cualquier otra entregada en cualquier convocatoria, tanto copiador como copiado quedarán inmediatamente suspensos hasta la siguiente convocatoria y sujetos a las medidas disciplinarias a las que hubiere lugar.

7. Plazos de entrega

El plazo para la primera entrega finaliza el día **16 de Diciembre de 2010** a las **23:59h**. El plazo para la segunda entrega finaliza el día **23 de Diciembre de 2010** a las **23:59h**.

[13] La signatura de un método consiste en su nombre, el número de argumentos de entrada, su orden y su tipo y el tipo devuelto por el método.