Universidad del Valle
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación
IPOO

Taller # 2
"Primera aproximación al diseño y construcción de clases en Java y C++".

Introducción
El objetivo de este taller es interiorizar en el estudiante la noción de clase como unidad de software. A la vez, se discutirán las semejanzas, así como algunas de las diferencias al momento de emplear dos de los lenguajes O.O. más populares a la fecha: C++ y Java.
El enfoque de este taller estará orientado al desarrollo de clases pequeñas, tanto en C++ como en Java.
Su fecha de entrega será discutida en clase y pueden participar hasta 2 (dos) estudiantes, como máximo, en un mismo grupo.
El fraude, entendido como la presentación de código fuente con alta similitud entre grupos diferentes, se penalizará con la nota de 0.0, para todos los grupos involucrados.

Cada grupo debe entregar, en el CampusVirtual, un archivo comprimido, llamado IPOO_Taller2.zip, que contenga:

- Lo solicitado en cada punto organizado en una carpeta independiente y cuyo nombre corresponda al punto abordado.
- Los códigos fuente. Estos archivos deben estar debidamente etiquetados, de acuerdo con lo discutido en clase, e incluir el nombre y código de cada miembro del grupo.
- Archivo de compilación `makefile` que compile todos los programas.
- Las imágenes generadas salvadas en un formato comprimido (PNG ó JPEG).

I. Interacción de aplicaciones mediante tuberías (pipelines) en el S.O.
El archivo adjunto `AplicacionEnCapas.zip` contiene dos clases implementadas en Java: `CapaEntrada.java` y `CapaSalida.java`, y un programa estilo C: `capaCpp.cpp`.
Observe que a pesar de que en las implementaciones en Java se observa la palabra reservada `class`, su estructura no dista mucho de la del programa estilo C. Esto se debe a que todas las declaraciones y líneas de programación pertenecen al ámbito del método `main`.
En principio cada capa (componente) funciona de manera independiente realizando una labor específica. Las diversas capas logran comunicarse entre sí, y simular ante el usuario el comportamiento de un único programa, gracias al redireccionamiento de los flujos de entrada y salida estándar realizado por el sistema operativo, y a que entre ellos hay un acuerdo (protocolo) de comunicación. `CapaEntrada` pregunta al usuario la operación que desea

realizar así como sus dos argumentos. `capaCpp` lee de la entrada estándar, realiza la operación correspondiente e imprime en la salida estándar el resultado respectivo. `CapaSalida` lee de la entrada estándar e imprime en una ventana gráfica el resultado. El comando para comunicar las aplicaciones es (en estilo Windows):

```
java CapaEntrada | capaCpp | java CapaSalida
```

II. Interacción de aplicaciones en Java y C++ Parte I [25 %]

Modifique las clases implementadas en Java de manera tal que su diseño corresponda realmente al paradigma de orientado a objetos, y ofrezca al usuario los métodos de la clase `Operaciones` (descrita en el siguiente punto).
Introduzca los comentarios correspondientes a todos los atributos y los métodos mediante tags ó etiquetas de documentación y genere las APIs correspondientes mediante el comando `javadoc`. Las clases modificadas deben implementar el método `public void interactuar( )` como único método público adicional al método constructor.
En una imagen represente los diagramas de clases empleando la notación discutida en clase.

III. Interacción de aplicaciones en Java y C++ Parte II [25 %]

Diseñe e implemente en C++ las clases `Operaciones` y `UsaOperaciones`. `Operaciones` implementa las cuatro operaciones básicas mediante el paso y retorno de parámetros. Es decir, no conoce a la clase `iostream` y por lo tanto no lee ni imprime datos. `UsaOperaciones` lee datos, envía mensajes y recibe datos de `Operaciones`, e imprime datos.
En dos imágenes:

- represente los diagramas de clases empelando la notación discutida en clase, y
- capture un screen-shot (pantallazo) de toda la aplicación donde se aprecien: el comando de ejecución, las dos ventanas y que se han realizado las cuatro operaciones.

III. Filtros en imágenes a color [50 %]

Implemente en C++ las clases `FiltrosColor` y `UsaFiltrosColor` de acuerdo a la siguiente declaración de clase. Comente todos los atributos y métodos, así como la clase empleando marcas de documentación.

```
class FiltrosColor
{
      private:
        unsigned int contador;
      public:
        FiltrosColor ();
        ~FiltrosColor();
        unsigned int getContador( ) const;

        bool histogramaAColor  ( Img * ptrImgOriginal, Img * ptrHistoColor);
        bool truncarAPisoImpar( Img * ptrImgOriginal, Img * ptrImgFiltrada);
        bool truncarA127 ( Img * ptrImgOriginal, Img * ptrImgFiltrada);
        bool quantizarA16 ( Img * ptrImgOriginal, Img * ptrImgFiltrada);
        bool diferenciaAbsoluta ( Img * ptrImgT0, Img * ptrImgT1, Img * ptrImgFiltrada);
        bool diferenciaPositiva ( Img * ptrImgT0, Img * ptrImgT1, Img * ptrImgFiltrada);
        bool diferenciaNegativa ( Img * ptrImgT0, Img * ptrImgT1, Img * ptrImgFiltrada);

};
```

```
class UsaFiltrosColor
{
      private:
        FiltrosColor * ptrFiltros;
        Img * imgInput1;
        Img * imgInput2;
        Img * imgOutput;
        string  cadIn1,cadIn2,cadOut;


      public:
        UsaFiltrosColor ();
        ~UsaFiltrosColor ();
        Void interactuar ();
};
```

La clase `FiltrosColor` aplica filtros de color pero no gestiona el manejo de memoria. Por su parte `UsaFiltrosColor`, interactúa con el usuario preguntándole qué imagen ó imágenes a color desea emplea, con qué nombre desea salvar la imagen resultante de aplicar el filtro (los nombres a usar no tienen espacios) y hace la gestión de memoria respectiva (pide y libera) en cada operación del usuario.

Para la ejecución de los filtros se adjuntan imágenes y pares de imágenes a color. Usted debe adjuntar en su informe de este punto una descripción que indique a qué imagen o pares de imágenes aplicó cada uno de los filtros, y la imagen resultante de la aplicación de los mismos.
La semántica de los métodos es la siguiente:

`histogramaAColor` Genera un histograma en el que cada canal RGB resultante corresponde al histograma del canal respectivo en la imagen original. Los puntos ocupados/encendidos en un canal se marcan como 255 y los libres/apagados se marcan con 0. Así pues, en la imagen resultante el color blanco denotará la presencia conjunta de los tres colores para una intensidad, y las variaciones en color denotarán diferentes relaciones. El número de intensidades se manejará de manera fija como 256 [0, 255].

`truncarAPisoImpar` Para intensidades pares (a excepción del 0) la ajusta al valor impar más cercano por debajo. Las intensidades impares las mantiene igual.

`truncarA127` Asigna a cada intensidad el valor piso de multiplicarla por el factor de 1/2. Implica un ajuste en los niveles de la imagen resultante.

`quantizarA16` Divide el intervalo [0, 255] en 16 intervalos de igual longitud causando una transformación de valores al intervalo [0,15]. Implica un ajuste en los niveles de la imagen resultante.

`diferenciaAbsoluta` Se aplica a un par de imágenes asignando a cada píxel de la imagen resultante la diferencia en valor absoluto entre pares de canales en las imágenes de entrada.

`diferenciaPositiva` Se aplica a un par de imágenes asignando a cada píxel de la imagen resultante la diferencia en valor absoluto entre pares de canales en las

imágenes de entrada, para los casos en que la intensidad en un canal de la primera imagen es mayor que su respectiva en la segunda imagen.

`diferenciaNegativa` Se aplica a un par de imágenes asignando a cada píxel de la imagen resultante la diferencia en valor absoluto entre pares de canales en las imágenes de entrada, para los casos en que la intensidad en un canal de la primera imagen es menor que su respectiva en la segunda imagen.

`getContador` Retorna el número de veces que el objeto ha aplicado un filtro.