Arquitectura Final
BICOTI-II

Marcelo Alarcón
Claudio Risso

20 de Mayo 2001

1 Introducción

En este último informe explicaremos en un nivel conceptual, la arquitectura implementada[1] en base a lo expuesto en los dos informes anteriores[Sod99][Sod00], con las modificaciones que surgieron durante el desarrollo, principalmente en busca de una mejora tanto en la *performance* como en la naturalidad de algunos procesos.
Comenzaremos con los componentes más sencillos y que mejor se ajustaron al análisis original, el módulo I/O; para proseguir con el modelo de visualización e interacción.
Es destacable aclarar que no existe al momento soporte para *3D*, aunque de todos modos se sugieren algunas posibles soluciones con el objetivo de servir de guía para futuros proyectos.

2 Modelo I/O

El módulo de I/O[2] ha sido sin duda el más estable desde el comienzo del proyecto. Su estructura actual esencialmente no ha cambiado y por lo tanto

[1] Este documento no pretende ser un manual de usuario, detallando todos los métodos implementados. Este detalle se hará en un documento posterior que esperamos se actualice periódicamente en la medida que se incremente el número de componentes.

[2] Recordar que en [Sod00] se había encontrado conveniente, limitar este módulo a manejo de dispositivos y formatos gráficos.

no nos extenderemos en justificar su conveniencia. A modo de repaso recordemos que se destacaban en este módulo dos clases nuevas: *bicotiDevice* y *bicotiMapper*.

2.1 Devices

El primero lidiaba con los detalles inherentes de los dispositivos. Presentaba una capa sobre la cual se interactúa con la abstracción *hardware*, funcionando como *drivers* que permiten operaciones de lectura y/o escritura. A este nivel se habla sencillamente en términos de *bytes*.

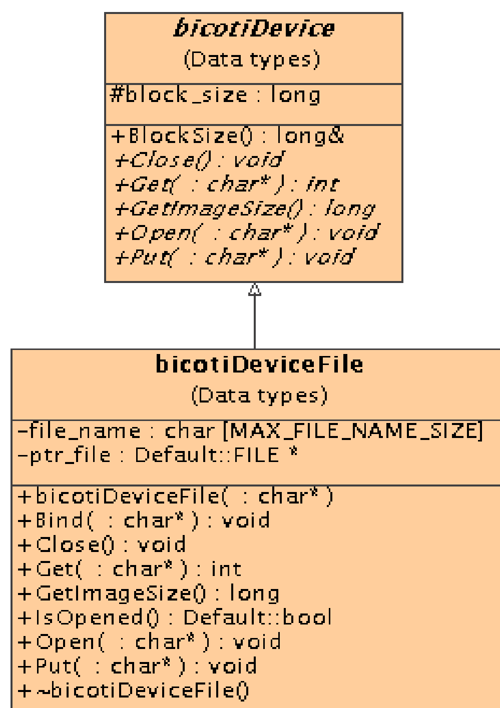

Figura 1: Los Dispositivos

2.2 Mapeadores

Los responsables de interpretar la información de los *devices* son los *mappers*. Estos últimos son la abstracción de los formatos gráficos y posibilitan esencialmente el almacenamiento y recuperación de imágenes. Como caso típico, un *mapper* asociado a un archivo(*bicotiDeviceFile*) y a una *bicotiImageImplementation[Val99]*, es responsable de interpretar la información del archivo y volcarla en forma consistente en la imagen.

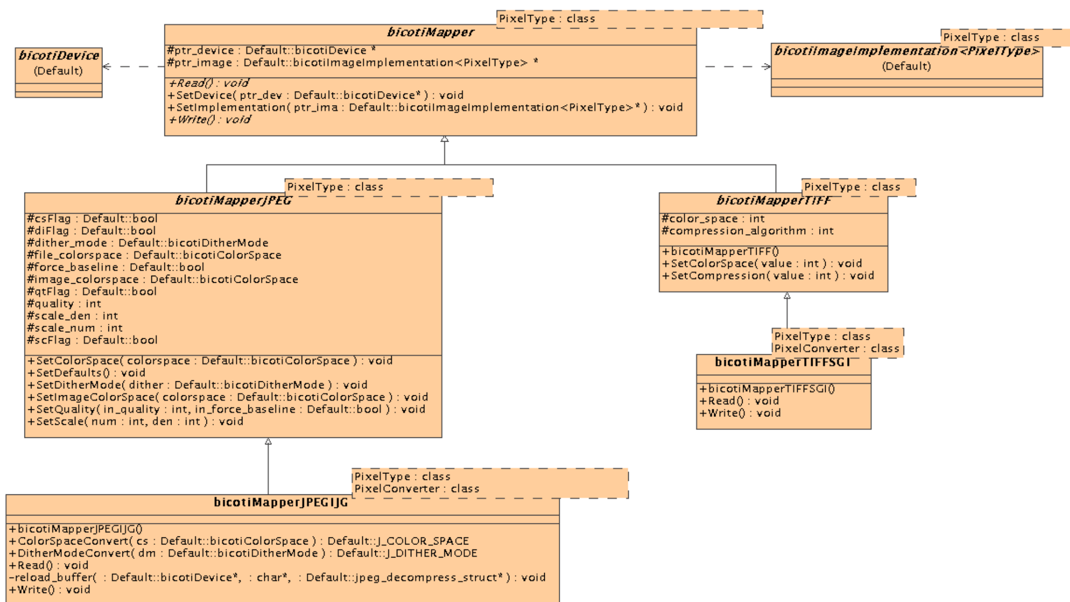

Figura 2: Los Mapeadores

En cuanto a la implementación, un problema adicional a considerar es la generalidad en el *pixel* de BICOTI-I , dado que en efecto, la mayoría (¿todos?) de los formatos actuales de imagen están limitados a un pixel *rgb*. Nuestro esfuerzo se centró en resolver el problema de importar imágenes hacia una imagen BICOTI. En algunos casos el proceso inverso puede significar pérdida de información[3] y seguramente se deberá definir en el futuro un formato propietario para algunos tipos de pixel no soportados convencionalmente.

[3]Pensemos en grabar una imagen de pixeles complejos en formato *jpeg*.

Para resolver el problema se ha re-utilizado el concepto de *bicotiPixelConverter*, basándonos en que de los formatos comunes es posible obtener *pixels RGBA*[4]. Como el mismo problema de generalidad se da en la visualización, no nos extenderemos ahora en los detalles de esta clase. De todos modos para fijar ideas, adelantaremos que cada implementación de esta clase nos permite convertir un formato de pixel a otro.
En **Fig.2** se observa que *bicotiMapper* es un *template*. Este hecho es inevitable debido a la presencia de *bicotiImageImplementation<PixelType>*. El esquema usado para *JPEG[IJG01]* y *TIFF[SGI01]*, así como el propuesto para incorporar un nuevo formato estándar (supongamos *BMP*), es escribir la especialización del mapper correspondiente (en este caso *bicotiMapperBMP*) basándose en *bicotiPixelRGBA* y usar como segundo parámetro del *template* a bicotiPixelConverterChar2RGBA, si se quiere escribir en una imagen monocromática. Este mecanismo que implica convertir uno por uno, todos los pixeles de la imagen a otro intermedio, no ha mostrado un pérdida importante de performance y da un interesante mecanismo para leer imágenes estándar, sobre cualquier tipo de *pixel*.

2.3 Ejemplo de I/O

```
#include <mapper.hpp>
#include <pixel_converter_rgba.hpp>

void main()
{
  bicotiImageImplementation2DArray< unsigned char > image_imp( 0 );
  bicotiDeviceFile archivo1( "entrada.jpg" );
  bicotiMapperJPEGIJG< unsigned char, bicotiPixelConverterChar2RGBA > mapper;
  mapper.SetDevice( &archivo1 );
  mapper.SetImplementation( &image_imp );
  mapper.Read();
  bicotiDeviceFile archivo2( "salida.jpg" );
  mapper.SetDevice( &archivo2 );
  mapper.Write();
};
```

[4]Rojo, Verde, Azul y un canal α de transparencia.

El ejemplo anterior muestra como leer una imagen en formato *jpeg* de disco (*entrada.jpg*), sobre una imagen de *pixel unsigned char* y grabarla nuevamente sobre el archivo *salida.jpg*. Esta claro que la información de color se ha perdido en el proceso. Una forma de evitar esto es usar *bicotiRGBA<unsigned char>* como pixel y *bicotiPixelConverterRGBChar2RGBA* como converter.

3 Visualización e Interacción

Por mucho el mayor desafío en este proyecto, estuvo justamente en este nivel. A diferencia de los algoritmos para formatos gráficos antes tratados, desarrollables como entidades internas a nuestra biblioteca o bien importables desde otras bibliotecas C++ estándar autónomas; los procesos de visualización e interacción se dan concurrentemente entre las distintas aplicaciones activas, a través de diversos, generalmente incompatibles *window managers* y bibliotecas gráficas. No es curioso por ende que sean pocas las bibliotecas para desarrollo de aplicaciones gráficas portables.
Desde el principio se ha buscado que BICOTI fuera una biblioteca *transparente* y *escalable[Sod99]* para el usuario, por lo tanto no se ha elegido el camino fácil que hubiera sido implementar en un biblioteca gráfica portable, un *widget* que permitiera mostrar una imagen. Esperamos como fruto de esta elección, integrar en el futuro *3D* sin que esto impacte en los desarrollos realizados hasta ese momento.

3.1 Interacción

Desde el análisis inicial se optó por el *modelo de delegación* para manejar los eventos en BICOTI. A continuación se introducen los conceptos generales y los detallan los componentes que fue necesario aislar e implementar.

3.1.1 Loop de eventos

Es común en todas las bibliotecas gráficas la existencia del llamado loop de eventos (*event loop*). La activación de este último implica que el programador desea detener el curso secuencial de la aplicación en ese punto, para transferir el control de las instrucciones al usuario.
En muchas aplicaciones la sucesión de operaciones está estáticamente establecida o está determinada por un conjunto información que puede asumirse

completamente conocido al comienzo de la aplicación. Leer una imagen de disco, calcular su histograma y verlo en la pantalla es un ejemplo de este tipo de aplicaciones en las cuales se puede prescindir de cualquier interacción. El *pseudocódigo* correspondiente sería:

```
Imagen imagen = LeerImagen( "archivo" )
Histograma histograma = CalcularHistograma( imagen )
MostrarHistograma( histograma )
```

En otras aplicaciones estas hipótesis no son válidas.

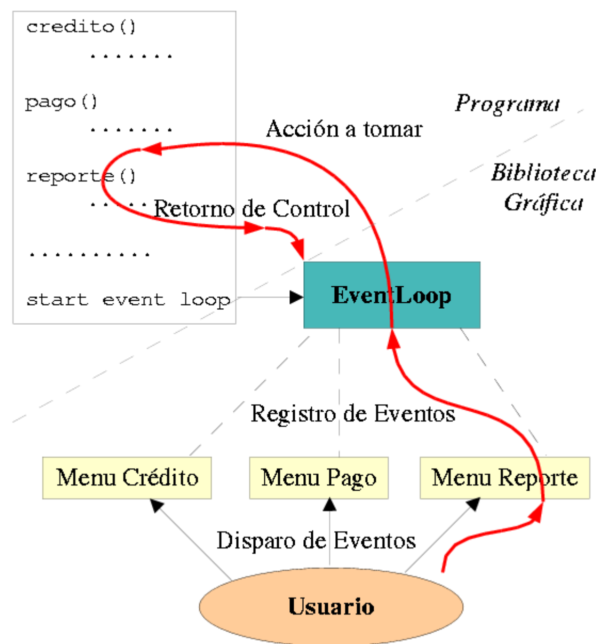

Figura 3: Event Loop

En un caso sencillo supongamos que en el sistema de control de pagos de una empresa existen tres funciones: *void credito(char * nombre, int monto)*, *void pago(char * nombre, int monto)* y *void reporte()* cuyas acciones modifican el estado de este de la siguiente forma:

- **credito.** Agrega en la base de datos de la empresa, un registro indicado que el cliente *nombre* a contraído una deuda por valor *monto* si es la

primera vez que este cliente compra a crédito en la empresa, o suma el monto nuevo al previamente adeudado si ya existía un registro de crédito para este cliente en la base de la empresa.

- **pago.** Descuenta del registro correspondiente al cliente *nombre* la cantidad *monto*.

- **reporte.** Genera un listado impreso de lo adeudado por cada cliente.

Como es evidente que la información en este caso no es conocida al momento de desarrollar el programa, un mecanismo para ingresar la misma mientras la aplicación esté activa es necesario. Es en este punto donde entra el loop de eventos que es el responsable de detectar que el usuario a elegido cierto componente (generalmente gráfico) que está asociado a alguna acción (una función). En este caso el control es transferido a la función mientras esta se ejecuta para volver una vez esta finalice al *event loop* como puede verse en forma esquemática en **Fig.3**. La ausencia del esquema secuencial clásico puede redundar en programas complejos y son necesarios mecanismos de alto nivel para facilitar la programación.

3.1.2 Los Actores

Aunque serán especificados más adelante, diremos que son actores todos las entidades capaces de disparar eventos y/o ser vistas gráficamente durante el desarrollo de una aplicación BICOTI. Son los actores los ladrillos con los que se construyen las aplicaciones gráficas y mediante los que se interactúa con la biblioteca gráfica particular que se está usando. Se desprende en consecuencia que son precisamente estos, las fuentes de eventos en el *modelo de delegación*.

3.1.3 Eventos reconocidos

El propósito de esta sección es enumerar los eventos identificados hasta el momento. En la mayoría de bibliotecas gráficas se reconoce un conjunto de eventos sensiblemente mayor al aquí identificado. Como posteriormente se verá, es relativamente sencillo agregar eventos nuevos en nuestra arquitectura y seguramente esto se hará incrementalmente a medida que surja la necesidad. Al igual que en BICOTI-I no se pretendió implementar todo algoritmo imaginable (entre otras razones porque hubiera sido en vano), sino dar un

framework sobre el cual crecer; en BICOTI-II el esfuerzo estuvo centrado en buscar una arquitectura con las mismas características.
Si bien son en definitiva los actores las fuentes de eventos concretas, la existencia de un módulo *interaction* donde se establece la naturaleza de los eventos, se justifica al ser la primer etapa a concretar a la hora de incrementar el grado de interacción. Este es primer módulo donde se deben agregar las clases que extiendan el grado de interacción soportado, sin necesidad aún de concentrarse en la implementación concreta de los mismos.
Los esquemas de interacción en cuestión son:

a. Interaction Timer: En este esquema hay dos tipos de entidades involucradas: *bicotiInteractionSourceTimer* que dispara *tics* periódicamente y *bicotiListenerTimer* que es notificado cada vez que se llega a una marca de tiempo en el *timer*. Como ocurre normalmente en el modelo, cuando un *listener*[5] está interesado en ser notificado de los eventos que sucedan en el *source*[6], debe declararse ante este último con el método *AddListenerTimer()*. En el momento que el *listener* no desee escuchar más los eventos debe eliminarse de la lista con *RemoveListener*. En **Fig.4** puede verse la relación explicada.

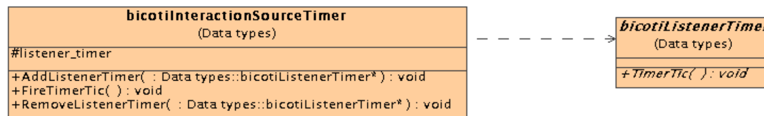

Figura 4: Interaction Timer

Se considera que un evento *TimerTic* ha sucedido, en el momento que es invocado el método *FireTimerTic()*.

b. Interaction Mouse: Un *bicotiInteractionSourceMouse* dispara a todos sus *listeners* un evento de tipo *MousePressed()* cuando se haya presionado el *mouse* o *MouseReleased()* cuando se haya liberado el mismo sobre el objeto[7]. En cualquier caso entre la información que es pro-

[5]En esta caso cualquier instancia de una clase que herede de *bicotiListenerTimer* e implemente el método *TimerTic()* para tomar la acción correspondiente.
[6]En esta caso una instancia de *bicotiInteractionSourceTimer*.
[7]A diferencia de a este objeto necesariamente es gráfico.

porcionada al *listener* están: el actor que está disparando[8] el evento en cuestión y una instancia de *bicotiEventMouse* con la información del caso. Esta información incluye: coordenadas (*GetX()* y *GetY()*) y botones(*IsLeftButton()*, *IsMiddleButton()*, *IsRightButton()*). El origen de coordenadas es la esquina superior izquierda del *source* con ejes horizontal a la derecha y vertical hacia abajo respectivamente. El diagrama de clases correspondiente puede verse en **Fig.5**.

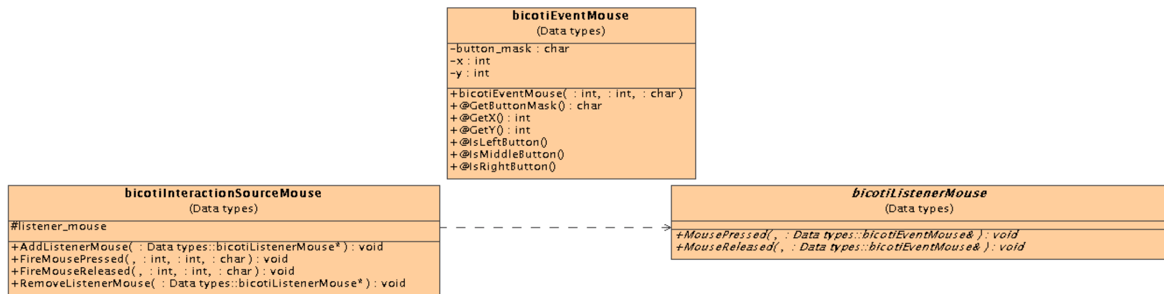

Figura 5: Interaction Mouse

c. <u>Interaction Mouse Motion:</u> Este tipo de interacción es muy similar a la anterior. De hecho podría haberse incluido en el mismo mecanismo si no fuera por una razón de conveniencia. Los eventos en este caso son disparados, cada vez que el mouse se mueve sobre el *source*. Estos eventos no se incluyeron en (b) por la pérdida de eficiencia que puede darse en el caso que un *listener* solo interesado en los *presseds* y *releaseds* sea bombardeado con los eventos de *mouse motion*. Por la misma razón se han incluido (*private*) los métodos *ActivateMouseMotionTracking()* y *DeactivateMouseMotionTracking()*. Ambos métodos son virtuales y están delegados al actor concreto. El primero indica que se dispare un *FireMoved()* cada vez que el mouse se mueva sobre el actor y es invocado cuando el primer *listener* se registra con *AddListener*. El segundo desactiva estos *fires* y es invocado cuando se hace el *RemoveListener*

[8]En todos los eventos esta información es más que conveniente para poder identificar la fuente si un *listener* está escuchando más de un *source*.

del último *listener* presente.

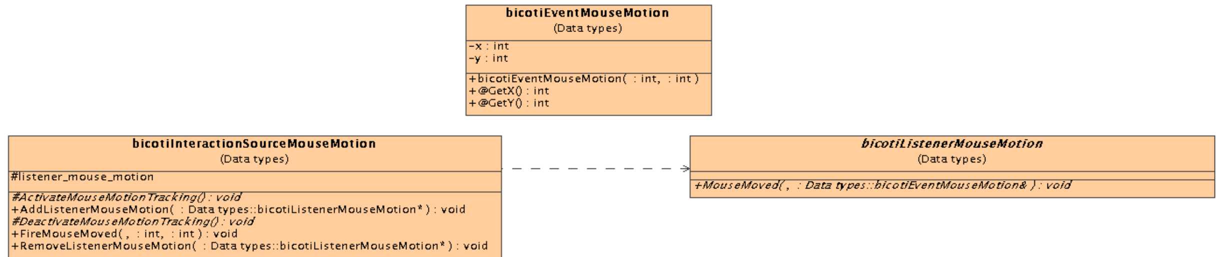

Figura 6: Interaction Mouse Motion

La anterior **Fig.6** esquematiza esta relación.

d. <u>Interaction Push Button:</u> Nuevamente tenemos eventos relacionados con el *mouse*. Ahora se genera un evento *clicked* cada vez que el botón izquierdo del *mouse* sea presionado y liberado sobre el actor. Este interacción en particular es programable a partir de (b), sin embargo la explicitación de la misma proporciona un mecanismo de alto nivel para reconocer este evento, típicamente asociado a los botones de una aplicación. En **Fig.7** se detallan los métodos involucrados.

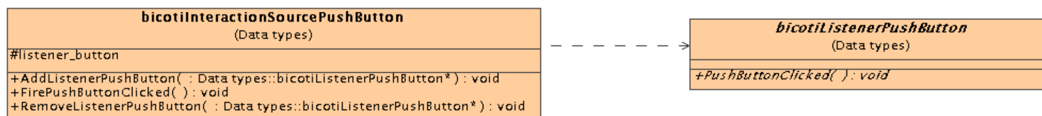

Figura 7: Interaction Push Button

e. <u>Interaction Menu:</u> Todo menú tiene uno o más *items* para seleccionar. Cuando un *bicotiListenerMenu* quiere estar al tanto si algún ítem es seleccionado, se registra en este con *AddListenerMenu()* suministrando el nombre del *item* en cuestión[9]. La figura **Fig.8** esquematiza la relación.

[9]En este momento se chequea la existencia del *item* con *CheckItem()*.

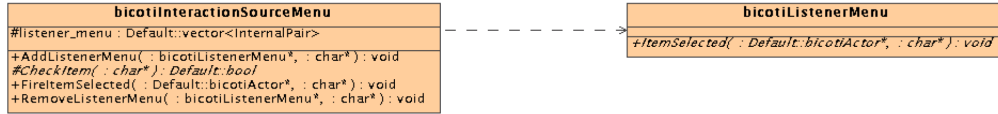

Figura 8: Interaction Menu

Los eventos: (a), (b), (c), (d) y (e) presentan como característica común no involucrar cambios de estado en el actor. A modo de ejemplo, el estado de un botón (posición, tamaño, texto, color, etc.) es independiente de que se produzca un *FirePushButtonClicked()* en este.
En cambio en (f), (g), (h) e (i), cualquier evento está asociado a un cambio de estado. Antes de entrar en detalles comentaremos la diferencia semántica entre un *fire* en ambos casos. En el primer grupo, un *fire* implica automáticamente que todos los *listeners* son notificados del evento. En el segundo en cambio hay que chequear que realmente se está dando un cambio de estado y cambiarlo efectivamente de ser cierto. Este control se delega a las clases especializadas (los actores), que son los que tienen la información de estado, mediante los métodos *check()* como se ve en el detalle de cada caso.

f. <u>Interaction Geometry:</u> Se entiende por evento geométrico, un cambio en la posición o en el tamaño en la fuente de eventos. Cuando se invoca un *FireActorMoved()* con la nueva posición del *source*, este último chequea que la posición a cambiado con el método virtual *CheckPosition*. Si el resultado de este es positivo, se procede a comunicar a todos los *listeners* disparando en estos el método *ActorMoved()*, con el parámetro *bicotiEventGeometry* que contiene la nueva posición del source. También es responsabilidad de *CheckPosition()* cambiar efectivamente la posición del actor en cuestión.
Un mecanismo completamente análogo se da en un cambio de tamaño como puede verse en **Fig.9**.

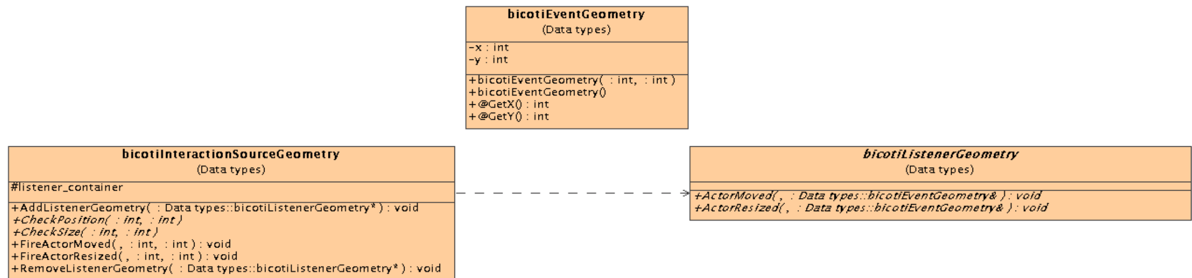

Figura 9: Interaction Geometry

g. Interaction Field: Un *text field* es un campo donde puede desplegarse un texto que puede ser editado interactivamente por el usuario. Los eventos interesantes para los *listeners* en este caso son: un cambio en el texto (*FieldTextChanged()*) y que se ha presionado el enter estando activa la edición (*FieldReturnPressed()*). Como el primer evento implica un cambio en el estado, existe un *CheckText()* asociado como puede apreciarse en **Fig.10**. Por el contrario no es necesario un método similar para *FireReturnPressed()*.

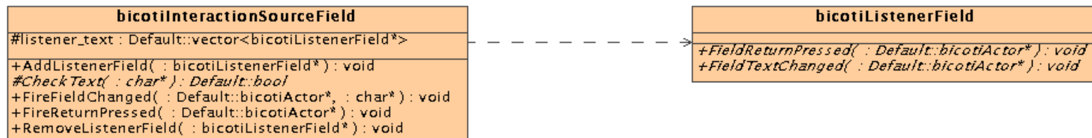

Figura 10: Interaction Text Field

h. Interaction Scroll Bar: El *scroll bar* en muchos de sus formas, es uno de los componentes más comunes en las aplicaciones gráficas. Aunque sus usos pueden ser muy diversos e ir desde desplazamientos de texto hasta selectores de valores; todos los *scroll bar* comparten una lógica común de interacción. Cada posición del *slider* está asociada a un valor en un rango de enteros, digamos [min, max]. Ciertos objetos (los *listeners* registrados) deben realizar acciones de acuerdo a este valor y por

lo tanto deben ser notificados de cualquier cambio en el mismo con el método *ScrollBarValueChanged*.

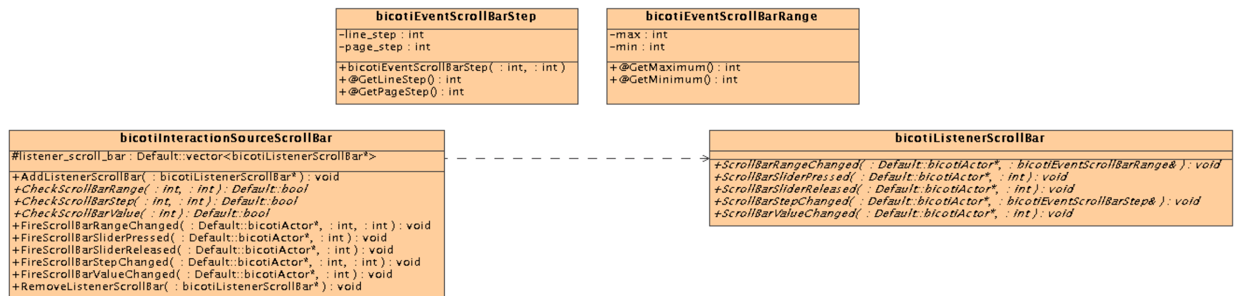

Figura 11: Interaction Scroll Bar

Suele existir más de una forma de cambiar la posición (valor) del *slider* entre las que rescatamos:

(i) Presionar las flechas en los extremos del mismo, lo que produce un incremento/decremento de cantidad *line step*.

(ii) Presionar el espacio entre la flecha y el *slider*, lo que produce un incremento/decremento de cantidad *page step*.

(iii) Presionar con el *mouse* el *slider* y moverlo hasta su nueva posición donde el *mouse* será liberado.

Con el fin de no tomar acciones frente a cada cambio de valor en el caso anterior, están discriminados los eventos *ScrollBarSliderPressed()* y *ScrollBarSliderReleased()*. Por último se registran los cambios en los parámetros del *slider*: *line step*, *page step* y el rango $[\min, \max]$. El resumen de los métodos se ve en **Fig.11**.

i. Interaction File Dialog: El *file dialog* es un *widget* de alto nivel que sirve típicamente para navegar en el disco duro hasta seleccionar un archivo en forma completamente gráfica. Los eventos destacables en este caso son: *FileDialogDirectoryChanged()*, *FileDialogFileHighlighted()* y *FileDialogFileSelected()* como se representan en **Fig.12**.

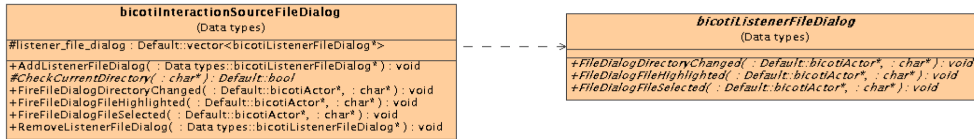

Figura 12: Interaction File Dialog

Aunque los nombres son bastante descriptivos comentaremos brevemente que:

(i) *FileDialogDirectoryChanged* es disparado cada vez que el usuario cambia de directorio durante la búsqueda de aquel que contiene el archivo deseado. Como el directorio actual de trabajo es parte del estado del actor, el *fire* correspondiente tiene asociado un *check*.

(ii) *FileDialogFileHighlighted* es disparado cada vez que el usuario *clickea* con el mouse un archivo en particular dentro del directorio de trabajo.

(iii) *FileDialogFileSelected* se aplica cuando se a elegido un archivo en particular. Esto último suele estar asociado a alguno de estos casos: se produce un doble-click en un archivo o luego de hacer un *highlight* se presionó el botón de *OK*.

3.1.4 Los Homólogos

El mecanismo elegido para lograr portabilidad en BICOTI-II , ha sido el encapsulamiento de otras bibliotecas gráficas. Cuando se desarrolla una aplicación, se hace enteramente con objetos de BICOTI. Antes de disparar el *event loop* de la biblioteca gráfica elegida, el usuario debe asegurarse de asociar cada actor de la aplicación con un objeto de la segunda biblioteca (el *homologo*) que será el efectivamente visible. Se han debido establecer en todos los casos mecanismos de comunicación entre estas entidades para ocultar al programador la existencia de los homólogos.

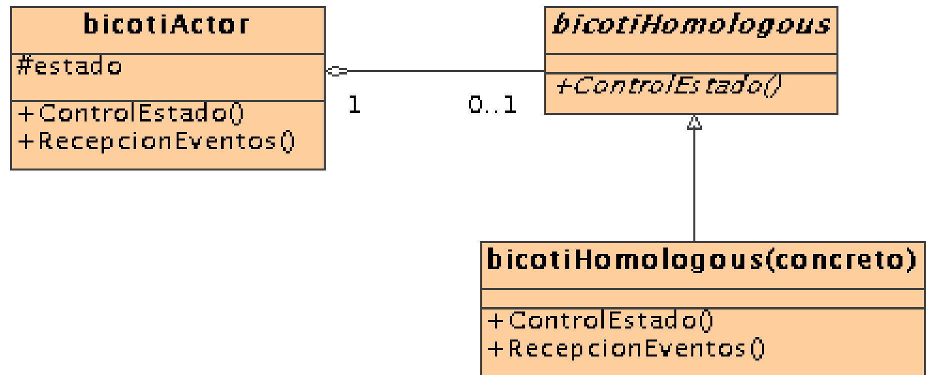

Figura 13: Relación Homólogo/Actor

Las implementaciones concretas se generan mediante la especialización de las interfaces abstractas definidas en el *bicotiHomologous* asociado al *bicotiActor*. La aplicación controla indirectamente al homólogo a través del actor y este a su vez dispara los eventos cuando es notificado de los mismos por el homólogo como puede verse en **Fig.13**. Durante la vida de un actor, puede haber momentos en que este tenga asociado un homólogo y otros en los que no; pero no puede existir un homólogo no asociado a un actor de ahí, la agregación entre ambos. Cabe destacar que un actor es siempre capaz de disparar eventos por *software*, tenga o no homólogo asociado. La existencia de este último agrega la posibilidad de disparar los eventos en forma interactiva. Algunos ejemplos de estos mecanismos se detallan en la siguiente sección.

3.1.5 Propagación de Eventos

En 3.1.3 fueron identificados varios eventos de interés. Estamos en condiciones ahora de ver el mecanismo completo de propagación de eventos y para fijar ideas se han elegido un par de casos representativos en sintonía con la categorización antes mencionada. Comenzaremos con un ejemplo sencillo en el cual el evento en cuestión no está asociado a un cambio de estado:

Ejemplo: *bicotiActorGraphic2DPushButton* - Explicaremos en detalle el funcionamiento de este actor y la relación con sus clases auxiliares. En

Fig.14 puede verse un esquema de todas las entidades involucradas en este proceso. Diferenciaremos dos situaciones que resumen el comportamiento:

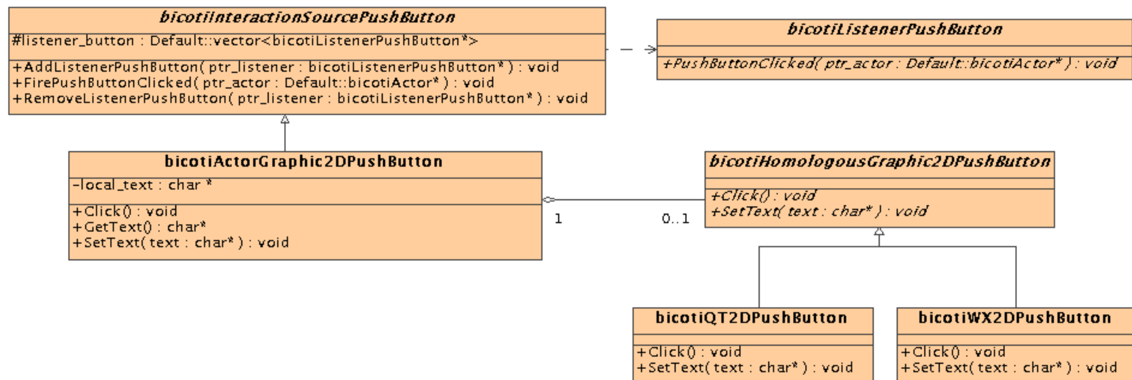

Figura 14: Funcionamiento del bicotiActorGraphic2DPushButton

En el primer caso un actor sin homólogo asociado recibe la orden *Click()* desde alguna parte de la aplicación y en consecuencia dispara el correspondiente *PushButtonClicked()* con un puntero a si mismo a cada uno de sus *listeners*.

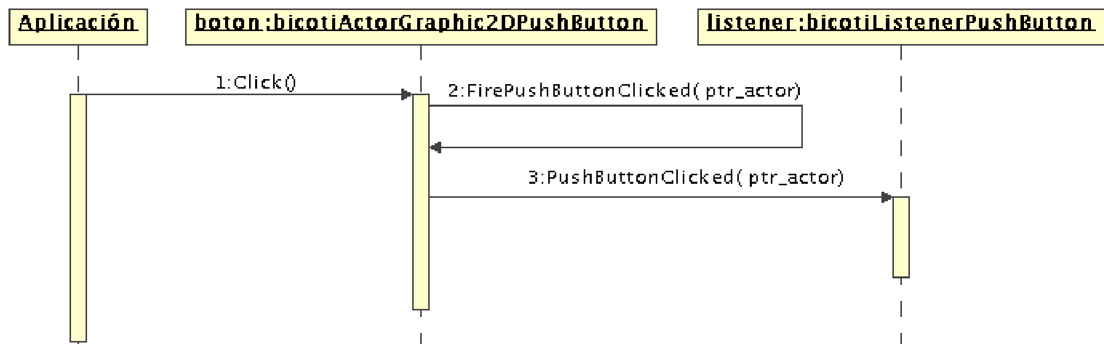

Figura 15: Click sin homólogo

Distinta es la situación en que existe el homólogo del actor, el *event loop* está activo y un usuario *clickea* el botón. En este caso el homólogo captura el evento de la biblioteca gráfica e indica al actor que dispare el *PushButtonClicked()*.

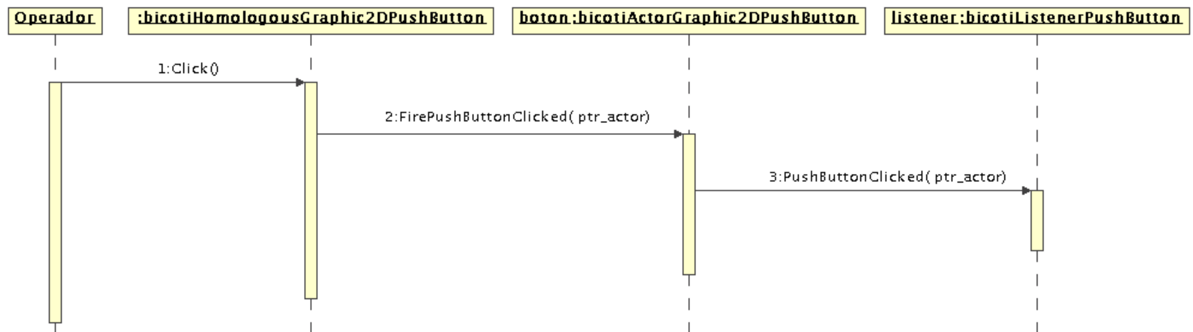

Figura 16: Click externo

Ejemplo: *bicotiActorGraphic2DField* - Explicaremos ahora un caso algo más complejo. A diferencia del ejemplo anterior, en un *Text Field*, la interacción puede cambiar el estado del actor.

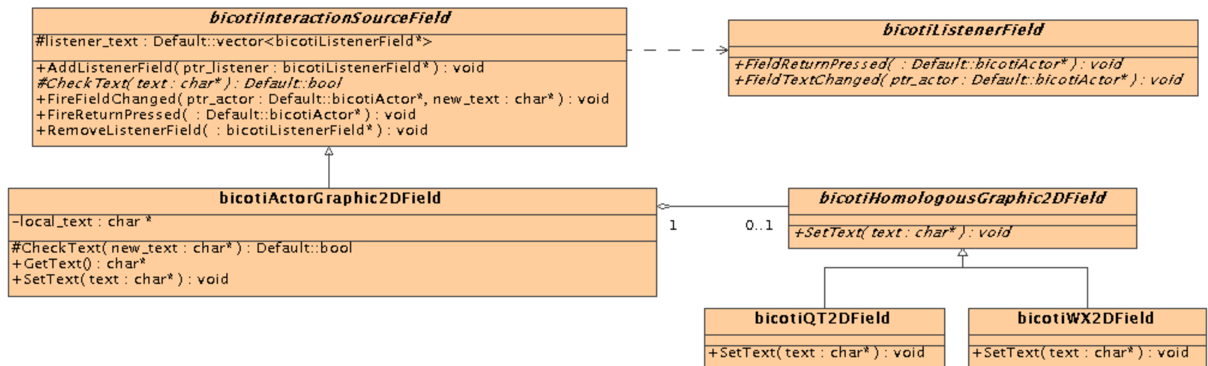

Figura 17: Funcionamiento del bicotiActorGraphic2DField

Son ahora tres las situaciones destacables:

Al igual que en el ejemplo previo comenzaremos por estudiar el caso en que el actor no tiene *homólogo* asociado y recibe de la aplicación un *SetText()*. Al no tener homólogo el *SetText()* invoca al *FireFieldChanged()* con un puntero a si mismo y el nuevo texto. Si este texto no coincide con el actual, se cambia efectivamente[10] y se notifica a todos los *listeners* del cambio.

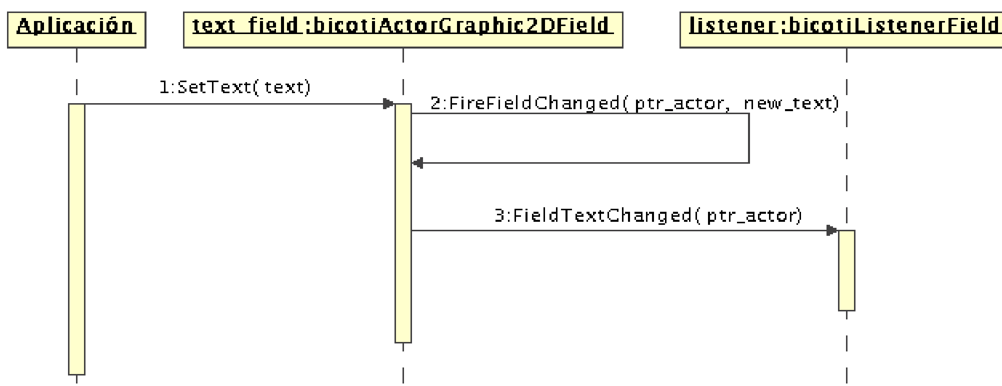

Figura 18: SetText sin homólogo

Nuevamente al igual que en el ejemplo previo, veremos la secuencia desencadenada por un operador que altera el texto sobre el homólogo de un *bicotiActorGraphic2DField*. En este caso, cada vez que el operador agrega un caracter al *text field*, se invoca un *FireFieldChanged()*, que al igual que en el caso anterior termina notificando a todos los *listeners* del cambio. Cabe destacar que aunque lo natural en este último caso parecería ser que el homólogo invocara un *SetText* en el actor, esto generaría una recursión infinita dado el comportamiento de este método, que se verá a continuación. Este diagrama de interacción puede verse en **Fig.19**.

[10]Recordar que esto es hecho por *CheckText()*.

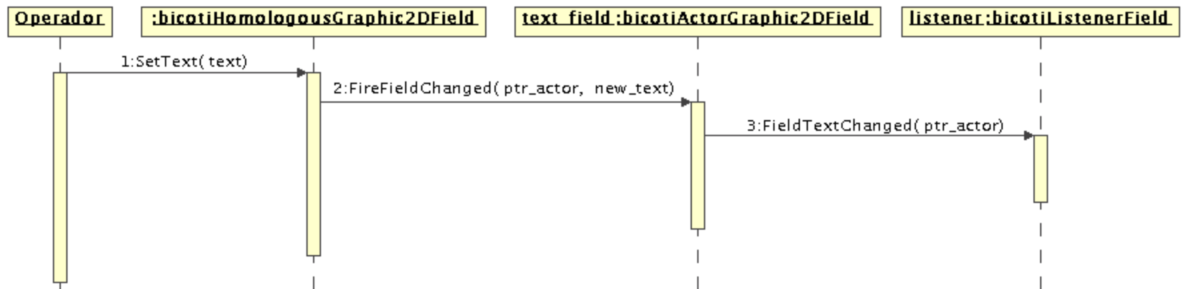

Figura 19: Cambio de texto externo

Solo resta ver que sucede si desde la aplicación, se invoca un *SetText()* sobre un actor con homólogo. Remarcamos en este punto que siempre es el actor el que guarda su información de estado, aún cuando tenga homólogo. Es importante por lo tanto que esta información esté siempre sincronizada. En este caso particular el texto debe cambiarse en ambos objetos (actor y homólogo) y eso es exactamente lo que sucede. En efecto el actor, al ver que tiene homólogo no dispara el *fire*, en su lugar redirecciona el *SetText()* al homólogo. Este recibe y procesa el evento del mismo modo que si hubiera venido del operador logrando el resultado deseado.

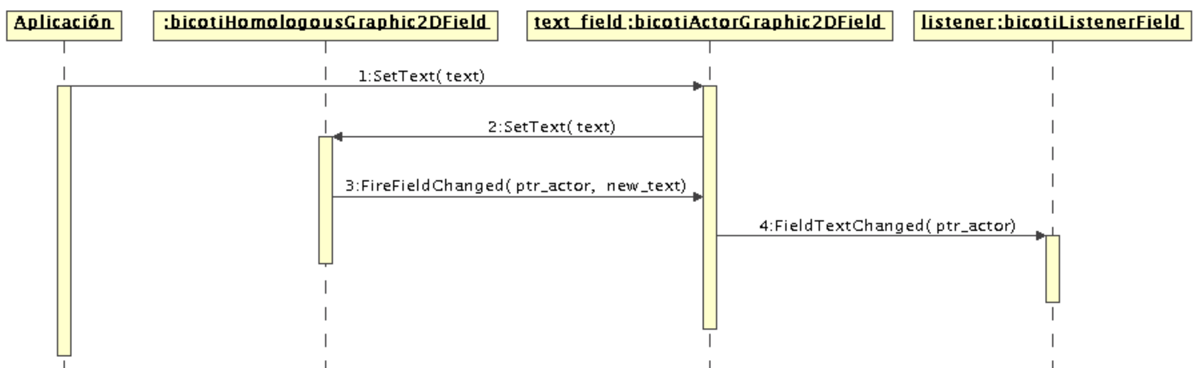

Figura 20: SetText con homólogo

3.1.6 El Homologous Factory

Si bien los homólogos encapsulan la biblioteca gráfica elegida, sería bastante tedioso el proceso de crearlos uno a uno, setear sus atributos y asociarlos con sus actores correspondientes. Más complejo aún sería migrar la aplicación a otra biblioteca gráfica cambiando todas las instancias de los homólogos. Un mecanismo de alto nivel para crearlos en forma transparente y asociarlos con los actores sería lo ideal. Más aún, es deseable poder cambiar de biblioteca en forma transparente en cualquier momento. Afortunadamente un mecanismo de tales características existe y es un conocido *pattern*; el *Abstract Factory*. El lector con experiencia en BICOTI-I , debería conocer otro *pattern* de esta familia; el *Factory Method* usado para construir los iteradores de cada implementación.
En aquel caso la interface de las implementaciones declaraba un método *CreateIterator()* que se implementaba en cada clase particular para retornar el iterador adecuado. Este mecanismo es posible gracias a la existencia de un único tipo de iterador asociado a cada implementación.
Marcando diferencias con el caso anterior destacamos la existencia de una gran variedad de tipos homólogos a asociar a un actor[11], lo que hace imposible usar el mismo esquema. El mecanismo en este caso, consiste en especializar una interface abstracta (el *bicotiHomologousFactory*), implementando el método *CreateHomologous()* que recibe un actor, usa los métodos introspectivos de este para identificarlo, crear y setear el homólogo adecuado.

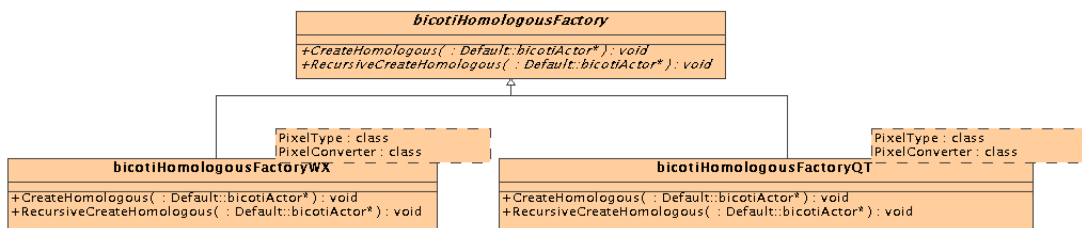

Figura 21: El *bicotiHomologousFactory*

Con este sencillo mecanismo, se consigue por ejemplo cambiar de *QT* a *WX Windows* cambiando un solo objeto; el *Factory*.

[11]De hecho habrá tantos posibles como bibliotecas gráficas soportadas.

3.2 Visualización

Haya o no interacción programada, es necesario lanzar el loop de eventos para ver los objetos gráficos. Los actores gráficos como el *Push Button* tienen el aspecto del *widget* asociado al homólogo en la biblioteca gráfica elegida y no merecen mayores comentarios. La novedad en la visualización que incorpora BICOTI-II es la referente a la posibilidad de ver imágenes BICOTI-I de pixeles genéricos. Esto es claramente una ventaja de BICOTI respecto a otras bibliotecas pero introduce un problema adicional de programación; ¿Como crear, procesar y ver imágenes con pixeles más genéricos que los soportados por las bibliotecas gráficas?. Más aún; ¿Como hacerlo en forma eficiente?.
Antes de ver la solución propuesta se impone un breve repaso de la implementación de BICOTI-I .

3.2.1 bicotiImageImplmentation

Como fuera en su momento documentado en [Val99], la *bicotiImageImplementation*, una de las clases centrales de BICOTI-I , es la responsable de almacenar la información de las imágenes digitales en su forma convencional; esto es como arreglos de pixeles.
Se hizo hincapié en dotar a la misma de un elevado grado de generalidad tanto en el pixel, como en la estructura usada para almacenar la información. El tipo de pixel a usar se parametrizó usando *templates* y distintas estructuras internas se logran mediante la especialización de las interfaces para cada dimensión. En la gran mayoría de los casos un *array* de pixeles es lo más adecuado para el procesamiento y se usaría la implementación *bicotiImageImplementation2DArray< PixelType >* para representarla. Pueden existir sin embargo otras imágenes en las que lo más adecuado sea almacenar solo los pixeles distintos de un valor por defecto y en este caso se elegiría la *bicotiImageImplementation2DSparse< PixelType >*. La estructura de esta familia puede verse en **Fig.22**.
Todos los algoritmos de BICOTI-I están escritos sobre las interfaces abstractas de las implementaciones y por lo tanto cualquier nueva implementación puede hacer uso de los mismos. Este hecho fue la base de la solución propuesta que se verá inmediatamente. Antes explicaremos como se atacó el problema de la generalidad en el pixel.

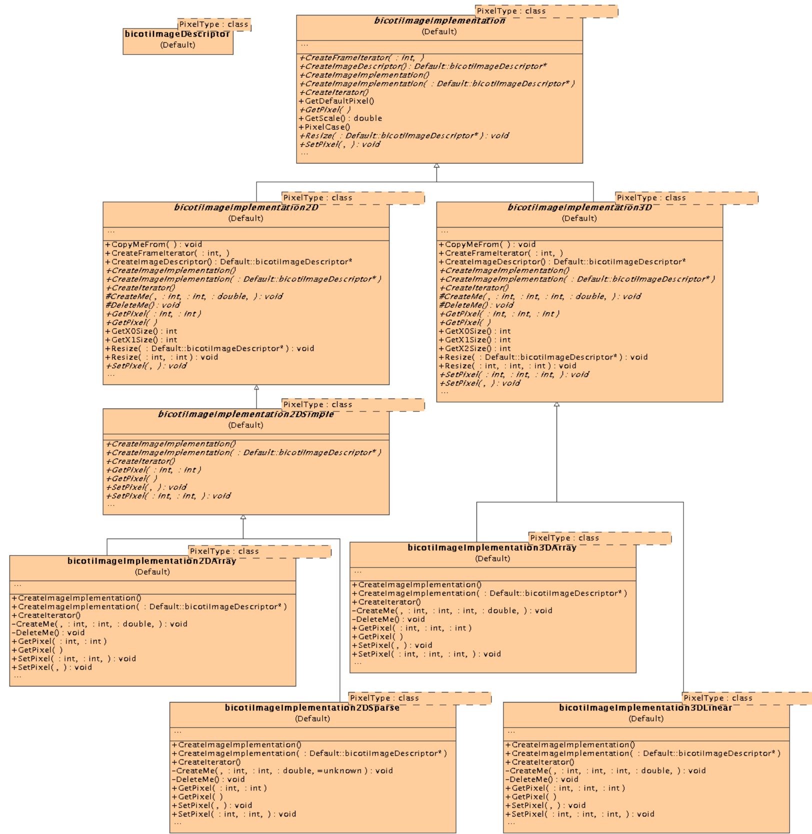

Figura 22: Las implemenaciones de imagen.

3.2.2 La necesidad del Pixel Converter

Ya habíamos visto como lidiar con la generalidad del pixel en (2.2) haciendo uso de los *PixelConverter*'s. Esta solución se mostró muy conveniente en su momento así que volveremos a usarla ahora con el propósito de unificar la forma en que BICOTI habla con el resto del mundo.
En los *mappers* se había adoptado el pixel RGBA como pixel central desde el cual se escribía la imagen en un pixel particular usando el *Pixel Converter* adecuado. Por ejemplo si la *bicotiImageImplementation* era monocromática con 256 niveles de gris, el tipo *unsigned char* era suficiente para el *PixelType*. Digamos que usamos *bicotiImageImplementation2DArray< unsigned char >* para fijar ideas. Necesitamos un *Pixel Converter* entonces capaz de

convertir entre pixeles RGBA y *unsigned char* y este sería el *bicotiPixelConverterChar2RGBA* de **Fig.23**.

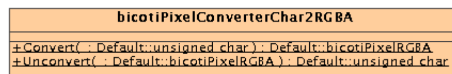

Figura 23: El *bicotiPixelConverterChar2RGBA*.

Si la imagen BICOTI usara color formato RGB con componentes en el rango $[0, 255]$, hubiéramos elegido un pixel *bicotiRGB*$<$ *unsigned char* $>$ y a *bicotiPixelConverterRGBChar2RGBA* para leer y escribir en archivos gráficos.

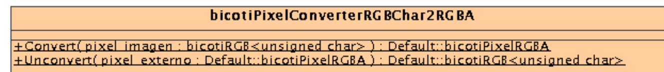

Figura 24: El *bicotiPixelConverterRGBChar2RGBA*.

En **Fig.25** se ve el esquema usado para los *mappers*. Para leer una imagen

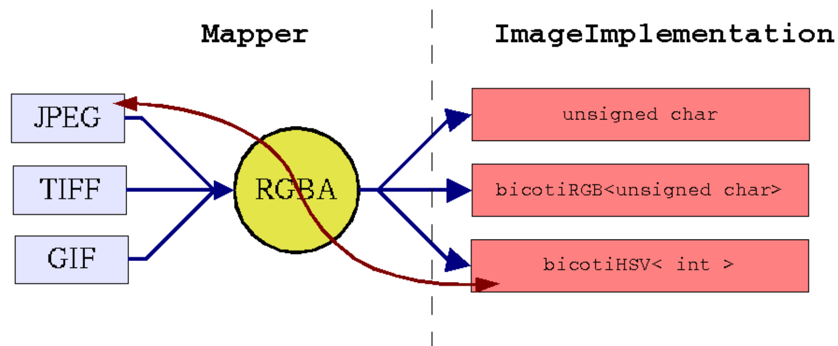

Figura 25: El *Pixel Converter* en los *mappers*.

desde un archivo en formato *jpeg* sobre una implementación BICOTI de pixeles HSV, el mapper es responsable luego de leer un pixel del archivo, de

pasarlo en formato RGBA al *PixelConverter* para que este los convierta al formato adecuado, en este caso *bicotiHSV< int >*.
Al ser clases paramétricas, los *Pixel Converter*'s no necesitan heredar de una interfaz común; solo es necesario que implementen como métodos estáticos: *Convert()* y *Unconvert()* con los parámetros adecuados al caso.
Para la visualización el esquema sería ligeramente distinto. Cada biblioteca gráfica tiene su propio pixel definido, y como es en estos donde hay que escribir en última instancia, habría inevitablemente que tener implementaciones de *Pixel Converter* para cada una. Adoptar el *bicotiPixelRGBA* como pixel intermedio implicaría dos conversiones para refrescar cada pixel de la implementación al homólogo. Si quisiéramos ver la imagen leída en el ejemplo anterior usando *QT* deberíamos usar nuevamente un *bicotiPixelConverterHSVInt2RGBA* para convertir cada pixel a RGBA y otro para pasar a *QRgb*, el pixel color de la biblioteca.

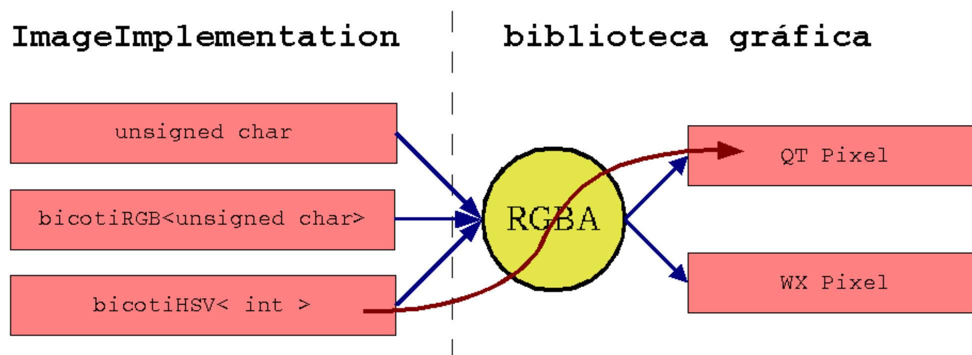

Figura 26: Una alternativa para la visualización.

En la visualización la performance es especialmente crítica y por lo tanto optamos por otro esquema en el que se suprime RGBA como pixel intermedio y se escribe directamente al pixel de la biblioteca gráfica. El esquema elegido puede verse en **Fig.27**. Para la misma función habría bastado un *bicotiPixelConverterHSVInt2QRGB*.

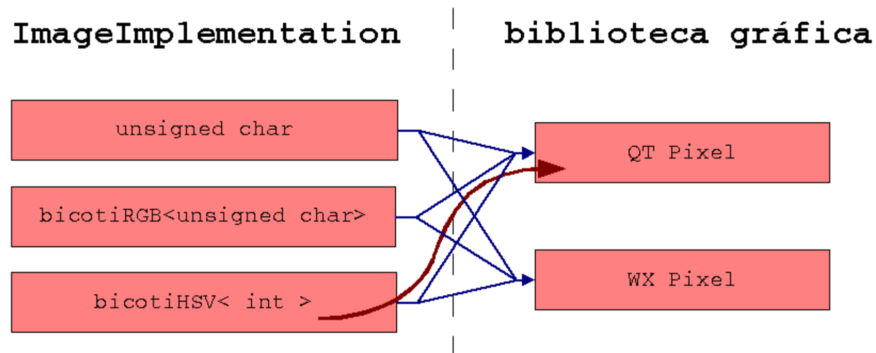

Figura 27: La alternativa elegida.

Para evaluar la conveniencia de la estrategia elegida comenzaremos viendo que perdemos respecto a la anterior:

(a). En primer lugar perdemos la posibilidad de re-usar lo conversores ya implementados para los *mappers*.

(b). Aumenta geométricamente el número de conversores al tener que implementar todas las combinaciones de estos.

(c). Se vuelve necesario que los *bicotiHomologousFactory* sean templates de *PixelConverter*. En consecuencia manejar simultáneamente más de un tipo de pixel en una misma aplicación, requiere la misma cantidad de *factories*.

Con la segunda alternativa se consigue una mejora en la velocidad de refresco de alrededor del 30%. ¿Vale este 30% las complicaciones acarreadas?. Estamos seguros que sí.

- Los puntos (a) y (b) no son tan críticos por la sencillez de los conversores, que realmente no suelen ser más de una par de lineas de código.
- El *factory* de todos modos iba a ser un *template* de pixel como veremos en (3.2.3) y en cualquier caso, la solución propuesta en (4) aligera el problema.

3.2.3 El Actor Image

La representación de imágenes está centrada en la *bicotiImageImplementation*. Todos los algoritmos implementados en BICOTI-I así como los *mappers* de BICOTI-II , se basan en esta y no es nuestra intención cambiarlo.
También es cierto que es indiscutible la necesidad de reconocer eventos sobre sobre su *widget* y por ende debería tener un actor asociado.
Ambos puntos estaban claro desde el principio del proyecto y en cierto modo ya fue discutido en los documentos previos [Sod99] y [Sod00]. Durante el desarrollo se exploraron diversas opciones que no serán expuestas aquí. Nos limitaremos a explicar la que fue en definitiva adoptada; el *bicotiActorImage*.
¿Que es el *bicotiActorImage*?... es un *bicotiActorGraphic2D* y una *bicotiImageImplementation2D* simultáneamente. En efecto, la generalidad en la implementación nos permite escribir una nueva que incluya métodos para manejar la interacción necesaria.

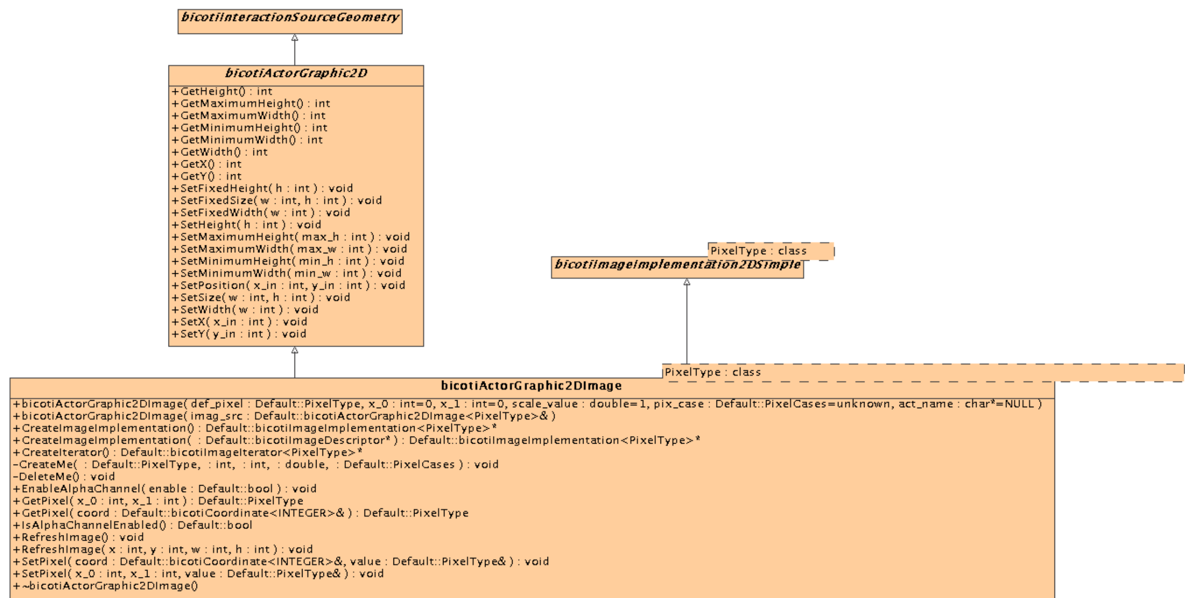

Figura 28: El *bicotiActorImage*.

Es inmediato que el *bicotiActorGraphic2DImage* es un *template* de pixel, así como su homólogo. Es inevitable que el *factory* lo sea y esto justifica

en parte la elección hecha en (3.2.2). El *Pixel Converter* está presente en el homólogo para de esta forma mantener la independencia de la biblioteca gráfica en el actor. Los algoritmos de procesamiento son aplicados directamente en el actor como en cualquier otra implementación y se refrescan en el homólogo por orden directa del usuario[12]. Aún cuando existen métodos para refrescar parcialmente la imagen en el actor, este homólogo incorpora otros adicionales para setear directamente pixeles a efectos de permitir una mejora de la performance en aplicaciones con alto grado de interacción.

3.3 Bibliotecas Gráficas

En (3.1.4) se introdujo el concepto de homólogo como encapsulamiento de las bibliotecas gráficas sobre las que se desarrolla BICOTI-II . En esta sección reseñaremos las distintas bibliotecas evaluadas y daremos un panorama más concreto sobre su desarrollo actual:

- **VTK**. VTK es una biblioteca de visualización orientada a imágenes 3D y portable a varias plataformas. Por estar el soporte 3D fuera del alcance del proyecto, se descartó esta biblioteca altamenente especializada en este tipo de imágenes.

- **QT**. QT es una biblioteca de componentes gráficos multiplataforma para el desarrollo de GUI. Su disponibilidad gratuita para Linux y abundante documetación la hicieron la candidata ideal para comenzar el desarrollo de BICOTI-II . La actual versión de BICOTI-II se implemetó sobre QT 1.44.

- **wxWindows**. wxWindows es una biblioteca gráfica multiplataforma para desarrollo GUI. Esta biblioteca fue propuesta por el IIE, básicamente motivados por:

  a. Ser gratuito en todas las plataformas (QT es un software comercial bajo MS Windows).

  b. Soporta la integración con VTK permitiendo reusar algoritmos desarrollados con esta herramienta.

[12]Cabe destacar que los refrescos pedidos por el *window manager* de turno son directamente resueltos por el homólogo.

Dado el estado bastante avanzado de la implementación de BICOTI-II con QT al momento de hacerse la propuesta, se siguió extendiendo el soporte sobre esta, haciendo énfasis en lograr un buen modelo, en el entendido que este simplificaría la incorporación en el futuro de otras bibliotecas, WX por ejemplo.

4 La Fachada de Actor Image

Los *templates* ofrecen muchas ventajas, especialmente desde el punto de vista de la *performance*. En contrapartida el uso excesivo de *templates*, sobre todos entre clases interdependientes, puede volverse tedioso por respetar la coherencia de tipos.
En BICOTI-II , no son muchos los *templates*; dentro de los actores, solo son templates los *bicotiActorGraphic2DImage*. Un imagen gráfica de pixel *unsigned char* sería una instancia de *bicotiActorGraphic2DImage<unsigned char>*. Un factory *QT* asociado sería *bicotiHomologousFactoryQT<unsigned char, bicotiPixelConverterChar2QRGB>*. El segundo parámetro de este último *template* debe ser compatible con el primero, es decir no son independientes.
El problema de la compatibilidad entre *templates* era mucho más grave en BICOTI-I y se había encontrado en su momento, una solución conveniente; la fachada. La fachada es un *pattern* cuyo propósito es enmascarar complejas relaciones entre diversos objetos detrás de una interface simplificada, que aún sin abarcar todo el dominio de aplicación enmascarado, logra prestar los servicios más generales del módulo.
Existe en BICOTI-I toda una arquitectura de fachadas, de hecho hay una fachada por módulo y fachadas que enmascaran otras fachadas, centrando todo en la *bicotiImageImplementation*. Nuestro equipo enriqueció las fachadas previas, agregando el módulo de *I/O* para todas las implementaciones y extendió la fachada de imagen para agregar capacidades gráficas en el caso específico del *Actor Image*.

Veamos otra versión del ejemplo (2.3), usando fachadas.

```
#include <image_2D_array_char.hpp>
void main( int argc, char ** argv )
{
  bicotiImage2DArrayChar imagen;
  imagen.BuildMapper();
  imagen.GetMapper()->BuildDeviceFile( "archivos1.jpg" );
  imagen.GetMapper()->BuildMapperJPEGIJG();
  imagen.GetMapper()->Read();
  imagen.GetMapper()->BuildDeviceFile( "salida1.jpg" );
  imagen.GetMapper()->Write();
  imagen.DestroyMapper();
};
```

Podemos ver la idea práctica de las fachadas. Cuando se elige una instancia de esta, se selecciona además de la dimensión e implementación, el tipo de pixel a usar[13] y el *template* para el resto de los objetos se resuelve automáticamente. La fachada encapsula además los punteros y las dependencias entre componentes.

En el siguiente ejemplo se usa la nueva fachada de *Actor Image* para leer la imagen de disco y mostrarla en una ventana del mismo tamaño de la imagen leída. Aunque a veces puede ser tedioso escribir los nombre de las clases BICOTI, son estos ejemplos casi autoexplicativos los que muestran la conveniencia en la elección. Merece comentario en el ejemplo la instrucción *SetMainActor(true)* del actor, indica que al cerrar la ventana se detenga el *Event Loop*.

```
#include <actor_image_2D_rgb_char_qt.hpp>

void main( int argc, char ** argv )
{
  bicotiActorImage2DRGBCharQT imagen;
  bicotiActorImage2DRGBCharQT :: EventLoopManager event_loop( argc, argv );
  bicotiActorImage2DRGBCharQT :: HomologousFactory factory;
```

[13]Debe existir la fachada implementada.

```
  imagen.BuildMapper();
  imagen.GetMapper()->BuildDeviceFile( "archivos.jpg" );
  imagen.GetMapper()->BuildMapperJPEGIJG();
  imagen.GetMapper()->Read();
  imagen.DestroyMapper();

  int width = imagen.GetImplementation()->GetX0Size();
  int height = imagen.GetImplementation()->GetX1Size();
  imagen.GetActor()->SetSize( width , height );
  imagen.GetActor()->SetMainActor( true );
  factory.CreateHomologous( imagen.GetActor() );

  event_loop.Start();
};
```

Terminaremos viendo otro ejemplo en el que leemos la imagen sobre un actor monocromático, le sumamos 1, calculamos el logaritmo y multiplicamos por 42 todos los pixeles de la imagen para finalmente mostrarlo.

```
#include <actor_image_2D_char_qt.hpp>

void main( int argc, char ** argv )
{
  bicotiActorImage2DCharQT imagen;
  bicotiActorImage2DCharQT :: EventLoopManager event_loop( argc, argv );
  bicotiActorImage2DCharQT :: HomologousFactory factory;

  imagen.BuildMapper();
  imagen.GetMapper()->BuildDeviceFile( "archivos.jpg" );
  imagen.GetMapper()->BuildMapperJPEGIJG();
  imagen.GetMapper()->Read();
  imagen.DestroyMapper();

  imagen.BuildOperatorUnary();
  imagen.GetOperatorUnary()->Addition( 1 );
  imagen.GetOperatorUnary()->Logarithm();
  imagen.GetOperatorUnary()->Product( 42 );
  imagen.DestroyOperatorUnary();
```

```
  int width = imagen.GetImplementation()->GetX0Size();
  int height = imagen.GetImplementation()->GetX1Size();
  imagen.GetActor()->SetSize( width , height );
  imagen.GetActor()->SetMainActor( true );
  factory.CreateHomologous( imagen.GetActor() );

  event_loop.Start();
};
```

Referencias

[IJG01] IJG. libjpeg reference. Technical report, Independent JPEG Group; URL: http://www.ijg.org, 2001.

[SGI01] SGI. libtiff reference. Technical report, The TIFF Official Site; URL: http://home.earthlink.net/ ritter/tiff, 2001.

[Sod99] Marcelo Alarcón; Claudio Risso; Carlos Soderguit. Análisis primario de requerimientos. bicoti-ii. Technical report, Facultad de Ingeniería, 1999.

[Sod00] Marcelo Alarcón; Claudio Risso; Carlos Soderguit. Definición de la arquitectura. bicoti-ii. Technical report, Facultad de Ingeniería, 2000.

[Val99] Claudio Risso; Rodolfo Rodríguez; Alvaro Valdes. Documentación de bicoti. Master's thesis, Facultad de Ingeniería, URL: http://www.iie.edu.uy/interno/g+i/bicoti.htm, 1999.