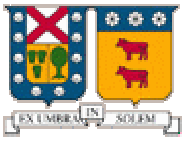
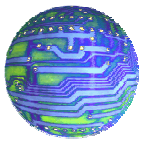


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA



Proyecto Final

Programación de Sistemas

"Java Beans"

Profesor: ***Agustín González V.***

Integrantes: ***Christian Lalanne A.***

Rodrigo Pinto A.

RESUMEN

A lo largo del presente proyecto se desarrollo ampliamente el tema de los Java Beans, en particular se han desarrollado (y explicado mediante distintos ejemplos) las bases para comenzar su utilización.

Lo primero que hacemos en este estudio hemos descargado BDK y hemos visto su contenido .Hemos aprendido a utilizar el BeanBox ,que sirve para desarrollar y probar componentes de software y hemos visto algunos ejemplos que BDK incluye .

Después estudiamos como funcionan los componentes de software y hemos examinado las clases e interfaces de los paquetes *java.beans* .

En el primer anexo introducimos el desarrollo de software basado en componentes y hemos estudiado y descrito como funciona el modelo de componente , y como los componentes de software se utilizan para simplificar el desarrollo de software complejo .Hemos estudiado el desarrollo de software basado en componentes y como javaBeans admite estas características.

En el segundo anexo hemos estudiado como escribir código de componentes de software en java.(por medio de ejemplos)

INTRODUCCIÓN

Anteriormente, todo el código se escribía línea por línea. Ahora, gracias a nuevas herramientas como Visual Basic, Java Beans, y otras, es posible ensamblar aplicaciones a partir de componentes prefabricados, apoyándose en código escrito anteriormente por otros desarrolladores de software. En general, estas nuevas tecnologías que permiten a los programadores construir entornos de programación visual, se conocen como ***ingeniería de software basado en componentes***. Así, y en vez de construir software desde el comienzo, escribiendo línea por línea el programa, podemos fabricar componentes de software u ocupar componentes creadas, para luego, ensamblarlos y crear un programa.

La tecnología de Java que nos permite construir componentes de software se llama Java Beans. Cuando la usamos, escribimos un pequeño componente funcional (un Bean), o usamos un Bean ya escrito para generar programas que de otra manera tendríamos que haber escrito en forma integra.

En el presente documento estudiaremos el modelo de componente de software Java Beans. Aprenderemos como funciona este y como se usan los componentes de software para simplificar el desarrollo de software más complejo.

Kit de Desarrollo Java Beans (BDK):

El kit de desarrollo Java Beans (BDK) esta disponible de forma gratuita en http://java.sun.com/products/javabeans/software/ ; Aquí se encuentra el BDK 1.1 disponible para bajar en formato zip, para su correcto uso, este kit de desarrollo necesita que el computador tenga instalado el SDK de Java ya que la herramienta esta completamente escrita en Java.

Instrucciones instalación para ambiente Windows:

- Descomprimir el archivo bajado en el disco fijo (por ejemplo C:\) con cualquier programa de descompresión (por ejemplo Winzip, http://www.winzip.com)
- El programa usado para la descompresión del archivo crea una carpeta llamada beans en el disco fijo seleccionado.
- Finalmente, para ejecutar el BeanBox (Herramienta de diseño visual), se debe cambiar al directorio C:\beans\beanbox y ejecutar el comando run

Instrucciones instalación para ambiente Linux:

- Descomprimir el archivo bajado en el disco fijo (por ejemplo C:\)con cualquier programa de descompresión (por ejemplo Winzip, http://www.winzip.com)
- El programa usado para la descompresión del archivo crea una carpeta llamada beans en el disco fijo seleccionado.
- Ejecutar el comando make ubicado en la carpeta beans creada por la descompresión
- Finalmente, entrar a la carpeta beanbox y ejecutar el comando.\run.sh

La aplicación BeanBox abre 3 ventanas principales (Toolbox, Beanbox y Properties) mostradas en las figuras a continuación:

- La ventana ToolBox contiene una lista de los componentes de software de Java que están a su disposición. Estos componentes se pueden usar para construir otros componentes más complejos, aplicaciones de Java o Applets

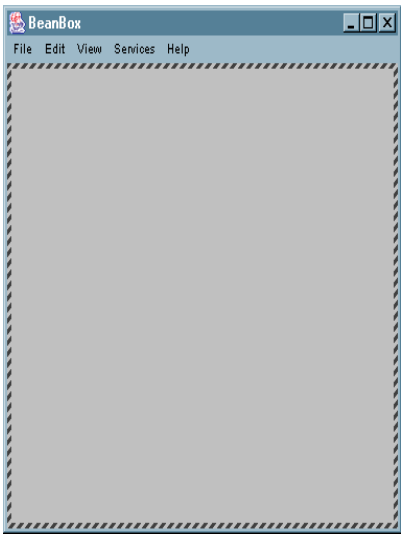


Figura 1

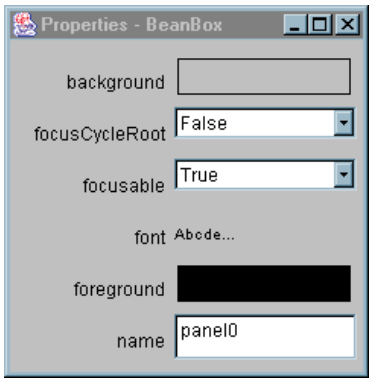


Figura 2

Figura 3

- La Herramienta de desarrollo visual de software, como BeanBox, permiten que los componentes de software estén visualmente organizados colocándolos en la ubicación deseada.

Uso del Bean Box:

Pulsamos en el componente Juggler (Malabarista) del Toolbox y luego pulsamos en la Beanbox; el componente Juggler se colocará en el Beanbox , como se muestra en la Figura 4.

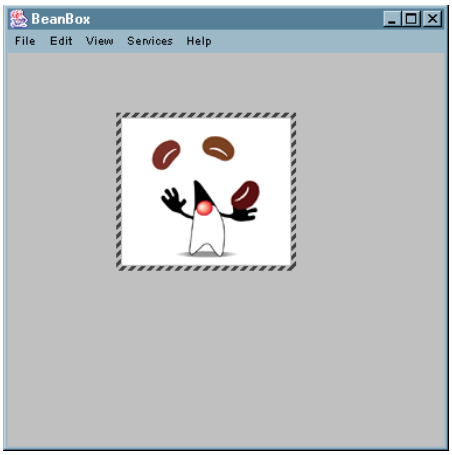


Figura 4

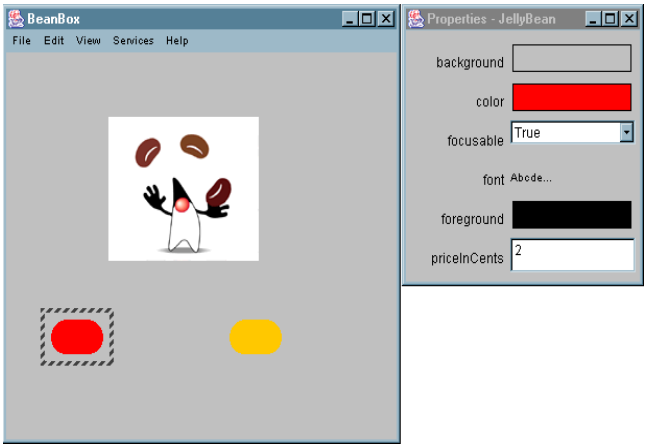


Figura 5

Observe que la ventana Properties se actualiza para mostrar las propiedades del componente Juggler. Puede personalizarse esta componente cambiando su propiedades. Por ejemplo Cambiamos la propiedad *animationrate a 1000,* entonces juggler disminuirá su velocidad.

Ahora añadimos 2 componentes jellybean de la Toolbox y los agregamos a la ventana Beanbox (podemos utilizar estos componentes jellybeans como botones), luego usamos la ventana *properties* para personalizar este jellybeans, por ejemplo, para cambiarlo de color (como vemos en la Figura 5).

Vamos a utilizar estos 2 jellybeans como botones para controlar la animación. Para esto conectaremos el manipulador de eventos del jellybeans como se muestra en la Figura 6 *mouseclicked*() del jellybean rojo al método *stopjuggling()* del componente juggler. Ahora pulse el jellybeans amarillo y posteriormente seleccione Edit | Events | mouse | mouseclicked de la barra de menú del beanbox. Ahora aparece una línea roja que precede al jellybeans amarillo. Esta línea representa una conexión lógica desde el manipulador de eventos del jellybeans. Ahora pulsamos el componente Juggler para cerrar la conexión. Al hacerlo, aparecerá el EventTargetDialogBox que se ilustra en la Figura 7. Este cuadro de diálogo lista los métodos de interfaz del componente Juggler. De estos, seleccionamos *startjuggling()* al hacerlo estaremos conectando el click sobre el jellybean al método *startjuggling()* a través del manipulador de eventos del jellybeans. El cuadro EventTargtDialog le notifica que esta compilando una clase adaptador. El Beanbox crea una clase especial, que se denomina clase *adaptador,* para conectar el

click del mouse sobre el jellybean con el método *startjuggling()* del juggler. Éste debe compilar esta clase y agregarla al Beanbox que esta corriendo para admitir esta conexión.

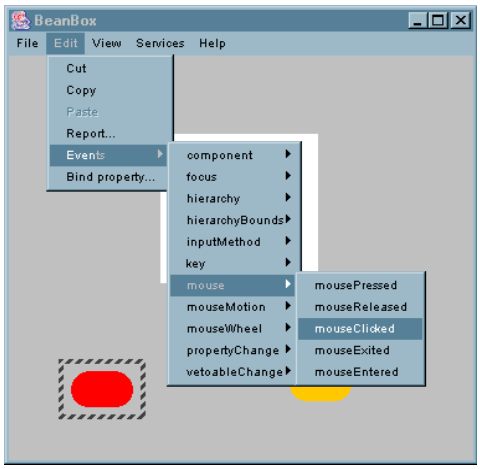


Figura 6

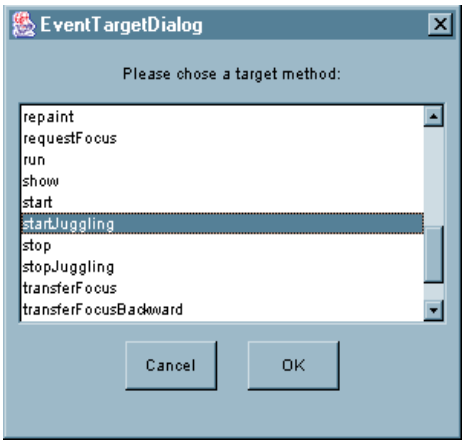


Figura 7

I.- Desarrollo de componentes de Software:

Algunos componentes de software son invisibles, en el sentido que carecen de despliegue gráfico. Las herramientas de desarrollo visual por regla general crean objetos gráficos especiales que permiten manipular los componentes de software invisible de la misma forma que se manipulan los visibles durante el desarrollo de software. Evidentemente estos objetos gráficos especiales de los componentes invisibles no aparecen en la aplicación o applet final.

Los componentes de software almacenan cualquier cambio en las propiedades, de forma que se ejecuten los valores nuevos de estas y se muestren cuando el componente que se haya modificado se use en una aplicación. La capacidad de almacenar permanentemente cambios en las propiedades se conoce como persistencia. Los componentes de software de Java implementan la persistencia serializando objetos componente que son ejemplos de una clase de componente. La serialización es el proceso de escribir en un flujo, el estado activo de un objeto. Dado que los componentes de software están serializados, deben implementar las interfaces Java.io.Serializable o java.io.Externalizable. Los componentes de software que implementan Java.io.Serializable se guardan automáticamente. Los componentes de software que implementan Java.io.Externalizable se guardan por sí mismos. Cuando un objeto componente se guarda a través de la serialización, también se guardan todos los valores de las variables del objeto. De esta manera, todo cambio en

las propiedades acompaña al objeto. Las únicas excepciones a esto son las variables que se identifican como *transient.* Los valores de las variables *transient* no están serializados.

II.- Propiedades de los componentes de Software:

En el ejemplo del *juggler* vimos ejemplos de propiedades simples. La propiedad *animationrate* del componente *juggler* utilizaba un valor numérico simple.

Una propiedad indexada es una propiedad que puede tomar un arreglo de valores. Este tipo de propiedades se emplea para controlar un grupo de valores relacionados que son del mismo tipo.

Una propiedad limitada es aquella que alerta a otros objetos cuando su valor cambia.

Una propiedad restringida es aquella en la que el objeto notificado puede confirmar o denegar el cambio.

III.- Métodos de Acceso:

Existen 2 tipos de métodos de acceso: obtención y establecimiento. Los nombres de los métodos de obtención empiezan por *get* y vienen seguido del nombre de la propiedad sobre la que se aplican. Los nombres de los métodos de establecimiento empiezan con *set* y vienen seguidos del nombre de la propiedad.

a.- Métodos utilizados con propiedades simples

Si un componente de software tiene una propiedad llamada *name* del tipo *nametype* que puede ser leída o escrita, deberá tener los siguientes métodos de acceso:

*Public nametype **get**name().*

*Public void **set**name (nametype namevalue).*

Una propiedad es de solo lectura o solo escritura si falta uno de los mencionados métodos de acceso.

b.- Métodos utilizados con propiedades indexadas

Un componente de software que tenga una propiedad indexada poseerá métodos que admitan lectura y escritura de elementos individuales del arreglo o del arreglo completo. Por ejemplo, si un componente de software tiene una propiedad *widget* indexada, en donde cada elemento del arreglo sea del tipo *widgettype*, tendrá los métodos de acceso siguientes:

*Public widgettype **get**widget(int index).*

*Public widgettype[] **get**widget().*

*Public void **set**widget(int index, widgettype widgetvalue).*

*Public void **set**widget(widgettype[] widgetvalues).*

c.- Métodos utilizados con propiedades limitadas

Los métodos que se utilizan con propiedades simples e indexadas que veíamos anteriormente se aplican también a las propiedades limitadas.

Las propiedades limitadas requieren que se notifique a ciertos objetos cuando estas experimentan un cambio. La notificación del cambio se realiza a través de un *PropertyChangeEvent.* Los objetos que deseen ser notificados del cambio en una propiedad limitada deberán registrarse como auditores. Así, el componente de software que este implementando la propiedad limitada suministrará métodos de esta forma:

Public void addPropertyChangeListener(PropertyChangeListener 1).

Public void removePropertyChangeListener(PropertyChangeListener 1).

La clase *PropertyChangeEvent* y la interfaz *PropertyChangeListener* se definen en el paquete Java.beans. Los métodos definidos anteriormente no identifican propiedades limitadas especificas. Para registrar auditores en el *PropertyChangeEvent* de una propiedad especifica se deben proporcionar los métodos siguientes.

Public void addPropertyNameListener(PropertyChangeListener 1).

Public void removePropertyNameListener(PropertyChangeListener 1).

En estos métodos, *PropertyName* se sustituye por el nombre de la propiedad limitada.

Los objetos que implementan la interfaz *PropertyChangeListener,* deben implementar el método *PropertyChange().* Este método lo invoca el componente de software para todos sus auditores registrados, con el fin de informarles de un cambio en una propiedad.

d.- Métodos utilizados con propiedades restringidas

Los métodos que se utilizan con propiedades simples e indexadas que veíamos anteriormente se aplican también a las propiedades restringidas. Además se ofrecen los siguientes métodos de registro de eventos:

Public void addVetoableChangeListener(VetoableChangeListener 1).

Public void removeVetoableChangeListener (VetoableChangeListener 1).

Public void addPropertyNameListener(VetoableChangeListener 1).

Public void removePropertyNameListener(VetoableChangeListener 1).

Los objetos que implementan la interfaz *VetoableChangeListener* deben implementar el método *vetoablechange()*. Este método lo invoca el componente de software para todos sus auditores registrados con el fin de informarles del cambio de una propiedad. Todo objeto que no apruebe el cambio en una propiedad puede arrojar un *PropertyVetoException* dentro del método *vetoablechange()* para informar al componente cuya propiedad restringida hubiera cambiado de que el cambio no se ha aprobado.

IV.- Introspección:

Con el fin de que los componentes de software sean utilizados por las herramientas de desarrollo visual, dichos componentes deben ser capaces de informar dinámicamente a las herramientas de sus métodos y propiedades de la interfaz, además del tipo de eventos que pueden generar o a los que pueden responder; esta capacidad se denomina introspección. La clase *Introspector* de java.beans proporciona un conjunto de métodos estáticos para que las herramientas obtengan información, métodos y eventos de un componente de software.

- Esquemas de reflexión y diseño: el paquete java.lang.reflect ofrece la posibilidad de identificar los campos y métodos de una clase. La clase *Introspector* utiliza esta posibilidad para repasar los nombres de los métodos de una clase de componentes, identifica las propiedades de un componente de software mirando los nombres de métodos de los esquemas de nombres de obtención y establecimiento. Identifica también la generación de eventos en un componente de software y las posibilidades de procesamiento buscando los métodos que siguen a las convenciones de nombres acerca de la generación y audición de eventos. Esto es en ausencia de información explícita.

- Especificación explicita: la información sobre una componente de software puede ser proporcionada (optativo) por una clase especial de información sobre componentes de software

que implementa la interfaz *BeanInfo*. Esta interfaz proporciona métodos para transportar explícitamente información sobre los métodos, propiedades y eventos de una componente de software. La clase *Introspector* reconoce las clases *BeanInfo* por su nombre. (por ejemplo, si una componente de software se ha implementado a través de la clase *math*, la clase *BeanInfo* relacionada se llamaría *mathBeanInfo*).

V.- Conexión de eventos a métodos de Interfaz:

Las componentes de software, al ser principalmente componentes GUI, generan y responden a eventos. Las herramientas de desarrollo visual ofrecen la posibilidad de unir los eventos generados por un componente de software con los métodos de manejo de eventos que han implementado otros componentes. El componente de software que genera el evento recibe el nombre de origen del evento. El componente de software que escucha y maneja el evento se denomina auditor de eventos.

Dentro de Java.beans

Las clases e interfaces de los paquetes Java.beans están organizadas en las categorías de: soporte de diseño, soporte de introspección y soporte de cambios en el manejo de eventos.

1.- Soporte de diseño

Las clases de esta categoría ayudan a las herramientas de desarrollo visual a utilizar componentes de software en un entorno de diseño. La clase beans proporciona 7 métodos estáticos que usan los constructores de aplicaciones:

- *Instantiate()* crea una instancia de un componente de software a partir de un objeto serializado.
- *isInstanceOf()* determina si un componente de software es de una clase o interfaz especificadas.
- *getInstanceOf()* devuelve un objeto que representa una vista determinada de un componente de software.
- *isDesignTime()* determina si los componentes de software están ejecutándose en un entorno de construcción de aplicaciones.
- *setDesignTime()* identifica el hecho de que los componentes de software están ejecutándose en un entorno de construcción de aplicación.

- *isGuiAvailable()* determina si una GUI esta a disposición de los componentes de software
- *setGuiAvailable()* identifica el hecho de que una GUI esta a disposición de los componentes de software.

La interfaz *Visibility* viene implementada por clases que admiten la posibilidad de responder a preguntas acerca de la disponibilidad de una GUI para un componente de software. Proporciona los métodos *avoidingGui(), dontUseGui(), needsGui()* y *okToUseGui()* La interfaz *VisibilityState* proporciona el método *isOkToUseGui()*.

Los métodos de la interfaz *PropertyEditor* vienen implementados por clases que admiten la edición personalizada de propiedades. Estos métodos admiten una gamma de editores de propiedades, simples y complejos. El método *setValue()* se usa para identificar el objeto que se va a modificar. El método *getValue()* devuelve el valor modificado. Los métodos *isPaintable()* y *printValue()* admiten la acción de pintar valores de propiedades en un objeto Graphics. El método *getJavaInitializationString()* devuelve una cadena de código java que se usa para inicializar el valor de una propiedad. Los métodos *setAsText()* y *getAsText()* se emplean para establecer y recuperar un valor de propiedad como objeto String. El método *getTags()* devuelve una arreglo de objetos String que son valores aceptables para una propiedad. El método *suportsCustomEditor()* devuelve un valor boolean que indica si el editor personalizado lo proporciona un *PropertyEditor.* El método *getCustomEditor()* devuelve un objeto que pertenece a una subclase de *Component* y que se usa como editor personalizado de las propiedades de un componente de software. Los métodos *addPropertyChangeListener()* y *removePropertyChangeListener()* se usan para registrar los manipuladores de eventos *PropertyChangeEvent* que esta asociado con una propiedad.

La clase *PropertyEditorManager* ofrece métodos estáticos que ayudan a los constructores de aplicaciones a localizar editores de propiedades para propiedades determinadas. El método *registerEditor()* se utiliza para registrar una clase de editor de una clase de propiedad determinada. Los métodos *getEditorSearchPath()* y *setEditorSearchPath()* admiten listas de nombres de paquetes que sirven para localizar editores de propiedades. El método *findEditor()* localiza un editor de propiedades de una clase determinada. Los editores de propiedades no registradas vienen identificados por el nombre de la propiedad seguido por el Editor.

La clase *PropertyEditorSupport* es una clase de utilidades que implementa la interfaz *PropertyEditor*. Los métodos de la interfaz *Customizer* vienen implementados por clases que proporcionan una interfaz gráfica que sirve para personalizar un componente de software . A estas clase se les exigen que sean subclases de *java.awt.Component* , de forma que puedan ser desplegadas en un panel .El método *addPropertyChangeListener()* se usa para activar un objeto que implemente la interfaz *PropertyChangeListener* como manipulador de eventos del *PropertyChangeEvent* del objeto que se esta personalizando. El método *removePropertyChangeListener()* se usa para borrar un *PorpertyChangeListener.* El método *setObject()* se usa para identificar el objeto que va a ser personalizado.

2.- Soporte de Introspección

- *La clase Introspector* proporciona métodos estáticos que usan los constructores de aplicaciones para obtener información acerca de la clase a la que pertenece un componente de software. El introspector reúne esta información por medio de datos que ofrece explícitamente el diseñador de componentes de software, a la vez que utiliza esquemas de reflexión y diseño cuando la información explicita no esta disponible. El método *getBeanInfo()* devuelve información de una clase como objeto *BeanInfo.* El método *getBeanInfoSearchPath()* devuelve un arreglo string que se utiliza como ruta de búsqueda para localizar clases *beaninfo.* El método *setBeanInfoSearchPath()* actualiza la lista de nombres de paquetes que se usan para localizar clases *BeanInfo.* El método *decapitalize()* se usa para convertir un objeto string en un nombre de variable estándar en lo que respecta a las mayúsculas.

- *La interfaz BeanInfo* los métodos de la interfaz *BeanInfo* los implementan clases que proporcionan información adicional sobre un componente de software. El método *getBeanDescriptor()* devuelve un objeto *BeanDescriptor* que suministra información acerca de un componente de software. El método *getIcon()* devuelve un objeto *Image* que se usa como un icono para representar un componente de software. Utiliza las constantes de icono

que define *BeanInfo* con el fin de determinar que tipo de icono debe devolverse . El método *getEventSetDescriptors()* devuelve un arreglo de objetos *EventSetDescriptor* que describe los eventos que genera un componente de software . El método *getDefaultEventIndex()* devuelve el índice del evento más utilizado de un componente de software. El método *getPropertyDescriptor* devuelve un arreglo de objetos *PropertyDescriptor* que admite la modificación de las propiedades de una componente de software .El método *getDefaultPropertyIndex()* devuelve la propiedad mas actualizada de un componente de software. El método *getMethodDescriptors()* devuelve un arreglo de objetos *MethodDescriptor* que describe los métodos a los que puede acceder externamente un componente de software .El método *getAdditionalBeanInfo()* devuelve un arreglo de objetos que implementan la interfaz *BeanInfo.*

-*La clase SimpleBeanInfo* proporciona una implementación predeterminada de la interfaz *BeanInfo.*

-*La clase FeatureDescriptor y sus subclases:* La clase *FeatureDescriptor* es la clase superior de la jerarquía de clases que usan los objetos *BeanInfo* para dar información a los constructores de aplicaciones. Proporciona métodos que utilizan sus subclases para reunir y dar información. La clase *BeanDescriptor* ofrece información general acerca de un componente de software , como la clase a la que pertenece el componente de software . La clase *EventSetDescriptor* ofrece información sobre los eventos que genera un componente de software. La clase *PropertyDescriptor* ofrece información sobre los métodos de acceso a una propiedad y el editor de propiedades .Viene ampliada por la clase *IndexPropertyDescriptor ,* la cual a su vez proporciona acceso al tipo de arreglo implementado como propiedad indexada e información acerca de los métodos de acceso a una propiedad.

Las clases *MethodDescriptor* y *ParameterDescriptor* proporcionan información sobre los métodos y parámetros de un componente de software.

3.- Soporte de cambio en el manejo de eventos

- El *PropertyChangeEvent* esta generado por componentes de software que implementan propiedades limitadas y restringidas como resultado del cambio en los valores de estas propiedades. La interfaz *PropertyChangeListener* viene implementada por las clases que

escuchan el *PropertyChangeEvent.* Se compone de un solo método, *propertyChange(),* que se utiliza para manejar el evento.

- La interfaz *VetoableChangeListener* viene implementada por las clases que manejan el *propertyChangeEvent* y arrojan un *VetoableChangeEvent* como respuesta a ciertos cambios en las propiedades. El método *VetoableChange()* se usa para manejar el *PropertyChangeEvent.*

- La clase *PropertyChangeSupport* es una clase de utilidades que puede venir subclasificada por componentes de software que implementen propiedades limitadas. Proporciona una implementación predeterminada de los métodos *addPropertyChangeListener(), removePropertyChangeListener()* y *fireVetoableChange().*

4.- Agregación

Esta interfaz, se usa para agregar varios objetos a un solo componente de software. La amplia la interfaz *Delegate*, que proporciona métodos para acceder a *Aggregate*. La clase *AggregateObject* es una clase abstracta que implementa la interfaz *Delegate* y que supone la base para la creación de otras clases de agregación. (La agregación es distinta de la herencia, solo es una forma de combinar múltiples objetos en un solo componente de software).

Anexo 1

“Características Principales de Java Beans”

(Enfoque Teórico)

Componentes y Contenedores:

Los componentes de software de Java están diseñados para una reutilización máxima. A menudo, son componente GUI que están visibles, pero también pueden tratarse de componentes algorítmicos invisibles.

Los componentes son entidades de software especializadas que pueden ser reproducidas, personalizadas e insertadas en aplicaciones y applets. Los contenedores son simplemente componentes que contienen otros componentes. Un contenedor se utiliza como estructura para organizar los componentes de una manera visual. Las herramientas de desarrollo visual permiten arrastrar y soltar, cambiar de tamaño y colocar los componentes en un contenedor. La interacción entre componentes se produce a través del manejo de eventos y la invocación de métodos. Casi todas las clases de objetos de java pueden implementarse como un Java Bean.

El punto principal que hay que recordar sobre los componentes y contenedores Java Beans es que admiten un enfoque de desarrollo jerárquico en el que se pueden ensamblar los componentes sencillos dentro de los contenedores para crear componentes mas complejos. Esta capacidad ofrece a los programadores de software la posibilidad de reutilizar al máximo los componentes de software a la hora de crear el software o mejorar el existente.

Introspección y Descubrimiento:

Las interfaces de componentes están bien definidas y pueden ser descubiertas durante la ejecución de un componente. Esta característica a la que se le denomina introspección, permite a las herramientas e programación visual arrastrar y soltar un componente en un diseño de un applet o aplicación y determinar dinámicamente que métodos de interfaz y propiedades de componente están disponibles. Los métodos de interfaz son métodos públicos de un componte de software están disponibles para que otros componentes los utilicen. Las propiedades son atributos de un componente de software y a las que se accede a través de métodos de acceso. Los Java Beans admiten introspección a múltiple nivel; a nivel bajo, por medio del paquete Java.lang.reflect (método de reflexión); este, permite que los objetos java descubran información acerca de los métodos públicos, campos y constructores de clase que se han cargado durante la

ejecución del programa. Todo lo que se tiene que hacer es declarar un método o variable como publico para que se pueda descubrir por medio de la reflexión.

Una de las posibilidades de nivel intermedio que ofrece Java Beans son los patrones de diseño. Los patrones de diseño son convenciones sobre nombres de métodos que usan las clases de introspección de Java.beans para deducir información sobre métodos de reflexión basados en sus nombres. Por ejemplo, las herramientas de diseño visual pueden usar los patrones de diseño para identificar una generación de eventos del componente de software y las posibilidades de procesamiento buscando métodos que cumplan con las convenciones de nombres sobre generación y audición de eventos. Las herramientas de diseño pueden utilizar los patrones para obtener mucha información de un componente de software si faltan datos explícitos.

En el nivel mas alto, Java Beans admite la introspección mediante el uso de las clases e interfaces que proporcionan información explicita acerca de los métodos, propiedades y eventos de los componentes de software. Proporcionando explícitamente dicha información a las herramientas de diseño visual, puede añadir información de ayuda y niveles adicionales de documentación de diseño, que serán reconocidos automáticamente y presentados en el entorno de diseño visual. Estas posibilidades son importantes porque permiten que los componentes de software se desarrollen de tal forma que la información sobre ellos la obtengan automáticamente las herramientas de diseño visual.

Métodos de Interfaz y Propiedades:

Las propiedades determinan el comportamiento de un componente. Las propiedades de un componente pueden ser modificadas durante el diseño visual de una aplicación. La mayoría de las herramientas de diseño visual proporcionan hojas que facilitan la configuración de estas propiedades. Las hojas de propiedades identifican todas las propiedades de una componente y a menudo ofrecen información de ayuda relacionada a estas propiedades. Estas propiedades y la información de ayuda las descubren las herramientas de diseño visual por medio de la introspección.

En Java Beans, se acceden a todas las propiedades a través de métodos de interfaz especiales, a los que se denomina métodos de acceso. Hay dos tipos: *Métodos de obtención* y

Métodos de Establecimiento. Los primeros recuperan los valores de las propiedades y los segundos los establecen. La mayoría de las herramientas de diseño visual ofrecen la posibilidad de conectar los eventos que se han generado en un componte por medio de los métodos de interfaz de otros componentes.

Persistencia:

Las hojas de propiedades de las herramientas de diseño visual se usan para hacer las propiedades de los componentes a medida de las aplicaciones especificas. Las propiedades modificadas quedan almacenadas de tal forma que permanecen junto al componente desde el diseño hasta la ejecución. La capacidad de almacenar los cambios de las propiedades de un componente se conoce como persistencia . La persistencia permite personalizar los componentes para su posterior uso.

Java Beans admite la persistencia a través de la serialización de objetos (capacidad de escribir un objeto java en un flujo, de tal manera que se conservan la definición y l estado actual del objeto). Cuando se lee un objeto serializado en un flujo, el objeto se inicializa exactamente en el mismo estado en el que estaba cuando fue escrito en el flujo.

Eventos:

Las herramientas de desarrollo visual permiten arrastrar y soltar, cambiar de tamaño y colocar los componentes en un contenedor. La naturaleza visual de esta herramienta simplifica en gran medida el desarrollo de interfaces de usuario, estas herramientas permiten describir de manera visual el manejo de los eventos de los componentes. El evento esta manejado por un manipulador de eventos. Los componentes de software pueden manejar los eventos que se produzcan de forma local. Los componentes de software también pueden dirigir una llamada a oros componentes para completar el manejo de un evento. Las herramientas d desarrollo visual admiten la conexión de fuentes de eventos con auditores de eventos a través de las herramientas de diseño grafico. En muchos casos, el manejo de eventos se puede ejecutar sin tener que escribir código de manejo de eventos, este código lo generan automáticamente las herramientas de diseño visual.

Diseño Visual:

Una de las grandes ventajas de usar un enfoque basado en componente para el desarrollo de software esta en que se puede utilizar herramientas de diseño visual. Estas herramientas simplifican en gran medida el proceso de desarrollo de software complejo. También le permiten desarrollar software de mejor calidad mas rápidamente y a un menor costo.

Anexo 2

“Ejemplos de Desarrollo de Componentes”

(Gauge, Tcanv yAppletQuiz)

En el presente Anexo, aprenderemos a crear componentes de software y utilizarlos en un applet. En primer lugar, crearemos un marcador (*gauge*) sencillo que se puede utilizar como dispositivo para applets y aplicaciones. A continuación, crearemos un componente de software que se pueda utilizar para mostrar texto sin el uso de un objeto *TextArea* o *TextField*. Después, aprenderemos a utilizar estos componentes en un applet que despliegue una serie de preguntas con opciones múltiples (Quiz).

Componente de Software *Gauge* (marcador):

Al revisar todas las secciones anteriores sobre la creación de un componente de software, podría haberse quedado con la impresión de que estos son complicados y difíciles de desarrollar, en realidad, esto no es cierto, ya que podemos convertir fácilmente las clases existentes en componentes de software con poco esfuerzo de programación.

A continuación, se muestra el código para un marcador sencillo. Este marcador se presenta como un cuadro tridimensional que se rellena entre sus valores mínimos y máximos. El color del borde del marcador y su color de relleno son ambos configurables. También lo son sus dimensiones y su orientación horizontal / vertical.

Gauge.java

```
import java.io.Serializable;
import java.beans.*;
import java.awt.*;
import java.awt.event.*;

public class Gauge extends Canvas implements Serializable {
// Establecer constantes y valores predeterminados
public static final int HORIZONTAL = 1;
public static final int VERTICAL = 2;
public static final int WIDTH = 100;
public static final int HEIGHT = 20;
public int orientation = HORIZONTAL;
public int width = WIDTH;
public int height = HEIGHT;
public double minValue = 0.0;
public double maxValue = 1.0;
public double currentValue = 0.0;
public Color gaugeColor = Color.lightGray;
```

```
public Color valueColor = Color.blue;
public Gauge() {
  super();
 }
public Dimension getPreferredSize() {
  return new Dimension(width,height);
 }
// Dibujar componente de software
public synchronized void paint(Graphics g) {
  g.setColor(gaugeColor);
  g.fill3DRect(0,0,width-1,height-1,false);
  int border=3;
  int innerHeight=height-2*border;
  int innerWidth=width-2*border;
  double scale=(double)(currentValue-minValue)/
   (double)(maxValue-minValue);
  int gaugeValue;
  g.setColor(valueColor);
  if(orientation==HORIZONTAL){
   gaugeValue=(int)((double)innerWidth*scale);
   g.fillRect(border,border,gaugeValue,innerHeight);
  }else{
   gaugeValue=(int)((double)innerHeight*scale);
   g.fillRect(border,border+(innerHeight-
gaugeValue),innerWidth,gaugeValue);
  }
 }
// Metodos para acceder a propiedades de componente de software
public double getCurrentValue(){
  return currentValue;
 }
public void setCurrentValue(double newCurrentValue){
  if(newCurrentValue>=minValue && newCurrentValue<=maxValue)
   currentValue=newCurrentValue;
 }
public double getMinValue(){
  return minValue;
 }
public void setMinValue(double newMinValue){
  if(newMinValue<=currentValue)
   minValue=newMinValue;
 }
public double getMaxValue(){
  return maxValue;
```

```
 }
public void setMaxValue(double newMaxValue){
  if(newMaxValue >= currentValue)
   maxValue=newMaxValue;
 }
public int getWidth(){
  return width;
 }
public void setWidth(int newWidth){
  if(newWidth > 0){
   width=newWidth;
   updateSize();
  }
 }
public int getHeight(){
  return height;
 }
public void setHeight(int newHeight){
  if(newHeight > 0){
   height=newHeight;
   updateSize();
  }
 }
public Color getGaugeColor(){
  return gaugeColor;
 }
public void setGaugeColor(Color newGaugeColor){
  gaugeColor=newGaugeColor;
 }
public Color getValueColor(){
  return valueColor;
 }
public void setValueColor(Color newValueColor){
  valueColor=newValueColor;
 }
public boolean isHorizontal(){
  if(orientation==HORIZONTAL) return true;
  else return false;
 }
public void setHorizontal(boolean newOrientation){
  if(newOrientation){
   if(orientation==VERTICAL) switchDimensions();
  }else{
   if(orientation==HORIZONTAL) switchDimensions();
```

```
    orientation=VERTICAL;
   }
   updateSize();
  }
  void switchDimensions(){
   int temp=width;
   width=height;
   height=temp;
  }
  void updateSize(){
   setSize(width,height);
   Container container=getParent();
   if(container!=null){
    container.invalidate();
    container.doLayout();
   }
  }
}
```

Para ejecutar este componente de software, copie el código anterior a un archivo de texto y llámelo *gauge.jar,* luego, copie este archivo a su directorio C:*beans**jars* y luego, inicie el BeanBox como se indica en la sección **Kit de Desarrollo Java Beans (BDK)**.

El BeanBox se abre y muestra las ventanas ToolBox, BeanBox yPropertySheet. Observará que hay un nuevo componente en el ToolBox, se trata del componente Gauge, observe que incorpora su propio icono como lo muestra la Figura 1.

Figura 1

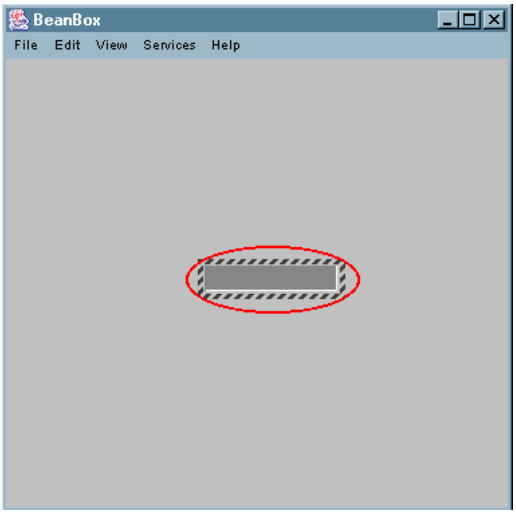


Figura 2

Pulse el icono del componente *gauge* en el ToolBox y luego pulse sobre el ToolBox, el componente aparece como un cuadro horizontal en 3D, como se muestra en la Figura 2.

La hoja de propiedades del componente de software muestra una serie de propiedades que se pueden variar para ver sus efectos en el componente recién agregado (Figura 3). Las propiedades *foreground, background* y *font*, son las propiedades predeterminadas de los componentes de software que están visibles. Estas propiedades reflejan los métodos *getForeground(), setForeground(), getBackground(), setBackground(), getFont()* y *setFont()* de la clase *Component.*

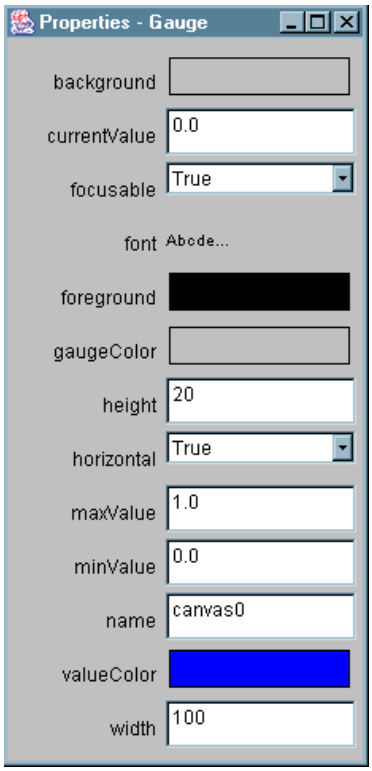


Figura 3

A continuación, se explican otras propiedades del componente de software *gauge*

- *minValue* y *maxValue*, identifican los valores mínimo y máximo que están asociados al marcador.
- *currentValue,* identifica el valor actual del marcador.
- *width y height,* controlan las dimensiones del marcador.

- *horizontal,* si esta seteado en True, el marcador aparece en forma horizontal, si esta seteado en False, aparece en sentido vertical.
- *gaugeColor y valueColor,* identifican el color del borde del marcador y el color que va a aparecer para identificar el valor actual del marcador.

Como funciona el componente de software Gauge:

Para empezar, observemos que en el código fuente del componente de software se importa *java.io.Serializable.* Todas las clases de componentes de software implementan *Serializable* o *Externalizable*, como se explico anteriormente; estas interfaces permiten la persistencia en los componentes, permitiendo que estos se lean y se escriban, almacenándose de forma permanente. Además de la serialización, no observara nada mas que sea excepcional en la clase *Gauge*. De hecho, se parece a cualquier otra clase personalizable del AWT.

El método *getPreferredSize(),* es un método que informa a las herramientas de construcción de la aplicación acerca de la cantidad de espacio que se necesita para mostrar un componente de software. Todos sus componentes visibles deben implementar *getPreferredSize()*.

El método *paint()* dibuja el componente de software en un objeto *Graphics*. Los componentes de software visibles necesitan implementar *paint()* con el fin de mostrarse a si mismos. El método *paint()* de *Gauge*, funciona dibujando un rectángulo en 3D por medio de gaugeColor y luego dibujando un rectángulo interno por medio de valueColor.

El marcador proporciona métodos de obtención y establecimiento para cada una de sus propiedades. Estos métodos se adhieren a las convenciones sobre nombres que se usan para las propiedades de los componentes de software. La clase *Introspector* de java.beans informa automáticamente de las propiedades que se corresponden con estos métodos a las herramientas de construcción de la aplicación, como es el caso de BeanBox.

El método *switchDimensions()* sirve para cambiar los valores de *width* y *height* cuando cambia la orientación del componente de software.

El método *updateSize()* se invoca cuando el componente cambia de tamaño. Invoca a *setSize()* para informar a un manipulador de diseño sobre su nuevo tamaño. Invoca al método *Invalidate()* de su contenedor para invalidar el diseño del contenedor y a *doLayout()* para hacer que el componente vuelva a aparecer.

La clase GaugeBean Info:

A continuación se detalla el código para que pueda desplegarse el icono del componente de software.

GaugeBeanInfo.java

```
import java.beans.*;
import java.awt.*;

public class GaugeBeanInfo extends SimpleBeanInfo {
// Devolver icono que se va a utilizar con el componente
 public Image getIcon(int size) {
  switch(size){
  case ICON_COLOR_16x16:
   return loadImage("gauge16c.gif");
  case ICON_COLOR_32x32:
   return loadImage("gauge32c.gif");
  case ICON_MONO_16x16:
   return loadImage("gauge16m.gif");
  case ICON_MONO_32x32:
   return loadImage("gauge32c.gif");
  }
  return null;
 }
}
```

La clase *GaugeBeanInfo* amplia la clase *SimpleBeanInfo* e implementa un solo método: *geticon()*. Este método lo invocan los constructores de aplicaciones con el fin de obtener un icono para un componente de software. Utiliza las constantes que se utilizan en la interfaz *BeanInfo* con el fin de seleccionar un color o icono monocromo de tamaño 16x16 ó 32x32 bits.

Componente de Software Lienzo de Texto:

Vamos a un componente de software con el cuál podremos dibujar texto en un lienzo, donde también podremos configurar sus fuentes y su métrica.

El nombre de esta componente de software será Tcanv y tendremos que copiar Tcanv.jar al directorio jars dentro de la carpeta Beans.

Cuando abramos nuestro Beanbox, veremos el componente de software TCanv en la ToolBox como se muestra en la Figura 4:

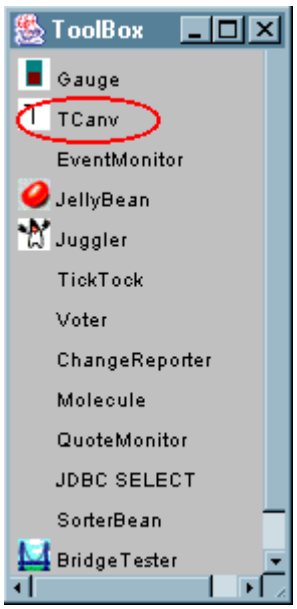


Figura 4

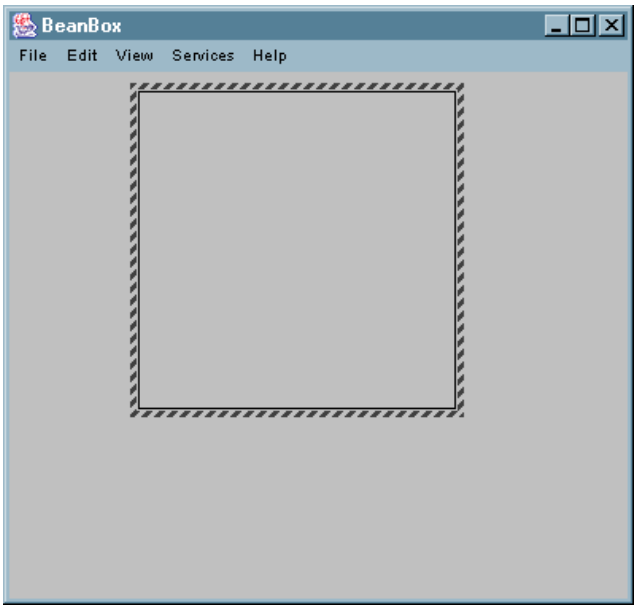


Figura 5

Pulse en el icono TCanv y luego en el Beanbox .Aparecerá el componente TCanv como se muestra en la Figura 5.

Las propiedades *background, foreground y font* son las propiedades predeterminadas de los componentes de software que están visibles. Las propiedades *leftMargin y topMargin* se usan para insertar espacio entre los extremos de un componente y el texto que este muestra. La propiedad *border* se usa para mostrar un borde alrededor del perímetro del componente de software. Las propiedades *widht y height* controlan las dimensiones del componente. La propiedad *text* identifica el texto activo que muestra el componente de software.

Haremos los siguientes cambios para ver como funcionan las propiedades del Tcanv:

- Cambie la propiedad *text* a Este|es|una|Prueba.
- Cambie las propiedades *topMargin* y *leftMargin* a 20.
- Cambie *font* a 14.
- Cambie la propiedad *background* a amarillo.

La Figura 6 muestra el efecto de estos cambios además de la ventana properties del TCanv donde se realizaron los cambios:

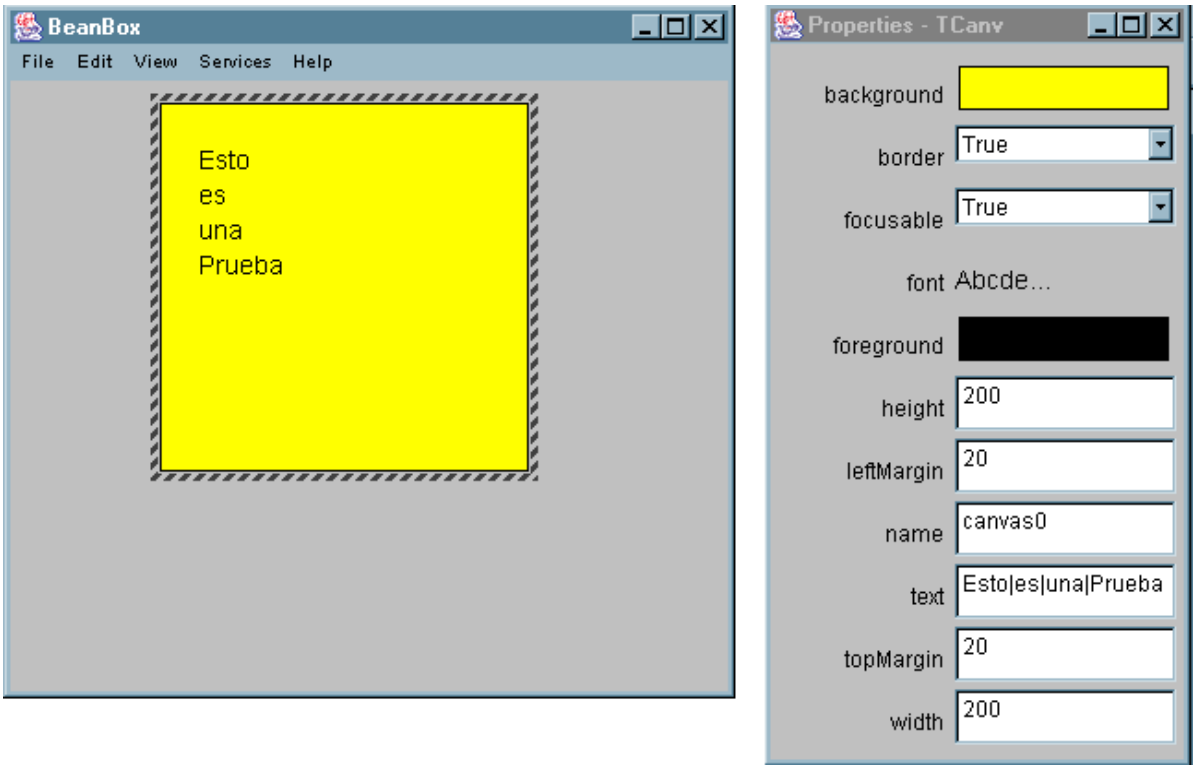


Figura 6

El código de este componente de software se muestra a continuación:

Tcanv.java

```
import java.io.*;
import java.util.*;
import java.beans.*;
import java.awt.*;
import java.awt.event.*;

public class TCanv extends Canvas implements Serializable {
 public static final int WIDTH = 200;
 public static final int HEIGHT = 200;
 public int width = WIDTH;
 public int height = HEIGHT;
 public int leftMargin = 5;
 public int topMargin = 5;
 public String text = "";
 public boolean border = true;
 public TCanv() {
  super();
```

```
}
public Dimension getPreferredSize() {
 return new Dimension(width,height);
}
public synchronized void paint(Graphics g) {
 if(border) g.drawRect(0,0,width-1,height-1);
 Font font = g.getFont();
 FontMetrics fm = g.getFontMetrics(font);
 int lineHeight = fm.getHeight();
 int y=fm.getLeading()+fm.getAscent();
 StringTokenizer tokenizer = new StringTokenizer(text,"|");
 String line;
 while(tokenizer.hasMoreTokens()){
  line=tokenizer.nextToken();
  if(border) g.drawString(line,leftMargin+1,topMargin+y+1);
  else g.drawString(line,leftMargin,topMargin+y);
  y+=lineHeight;
 }
}
public String getText(){
 return text;
}
public void setText(String newTextValue){
 text=newTextValue;
}
public int getWidth(){
 return width;
}
public void setWidth(int newWidth){
 if(newWidth > 0){
  width=newWidth;
  updateSize();
 }
}
public int getHeight(){
 return height;
}
public void setHeight(int newHeight){
 if(newHeight > 0){
  height=newHeight;
  updateSize();
 }
}
public int getLeftMargin(){
```

```
  return leftMargin;
 }
 public void setLeftMargin(int newLeftMargin){
  if(newLeftMargin >= 0) leftMargin=newLeftMargin;
 }
 public int getTopMargin(){
  return topMargin;
 }
 public void setTopMargin(int newTopMargin){
  if(newTopMargin >= 0) topMargin=newTopMargin;
 }
 public boolean isBorder(){
  return border;
 }
 public void setBorder(boolean newBorder){
  border = newBorder;
 }
 void updateSize(){
  setSize(width,height);
  Container container=getParent();
  if(container!=null){
   container.invalidate();
   container.doLayout();
  }
 }
}
```

La clase Tcanv, al igual que la clase Gauge, amplía Canvas e implementa *Serializable.* Define las variables de campo que corresponden a sus propiedades e implementa *getPreferredSize()* y *paint()*. También implementa algunos métodos de obtención y de establecimiento.

El método *paint()* comprueba la variable *border* y dibuja un borde (en caso de necesidad) alrededor del componente de software. Luego obtiene el valor de la *font* activa y el objeto *FontMetrics* de la fuente. Invoca al método *getHeight()* de la clase *FontMetrics* para obtener el alto de línea de la fuente activa en píxeles. Después utiliza un objeto *StringTokenizer* para analizar sintácticamente al objeto *String* de la variable *text* en base al delimitador |. Por último, el texto aparece una línea cada vez. Los métodos *hasMoreTokens()* y *nextToken()* de

StringTokenizer se usan para entrar en la cadena *text* y para que aparezcan en el objeto *Graphics* del lienzo del componente de software.

El siguiente código se usa para proporcionar íconos a los constructores de aplicaciones.

TcanvBeanInfo.java

```
import java.beans.*;
import java.awt.*;

public class TCanvBeanInfo extends SimpleBeanInfo {
 public Image getIcon(int size) {
  switch(size){
  case ICON_COLOR_16x16:
   return loadImage("tcanv16c.gif");
  case ICON_COLOR_32x32:
   return loadImage("tcanv32c.gif");
  case ICON_MONO_16x16:
   return loadImage("tcanv16m.gif");
  case ICON_MONO_32x32:
   return loadImage("tcanv32c.gif");
  }
  return null;
 }
}
```

El Applet Quiz:

Ahora que ya tenemos el componente de software *gauge* y *Tcanv* los utilizaremos en un applet. Este applet utiliza ambos componentes de software .Muestra al usuario una serie de preguntas aritméticas de opciones múltiples. Estas preguntas se muestran en una componente de Software *Tcanv* .Un segundo componente *Tcanv* muestra información sobre el estado .Un componente *Gauge* muestra el resultados de las respuestas en forma gráfica. La Figura 7 muestra como se presenta inicialmente el applet.

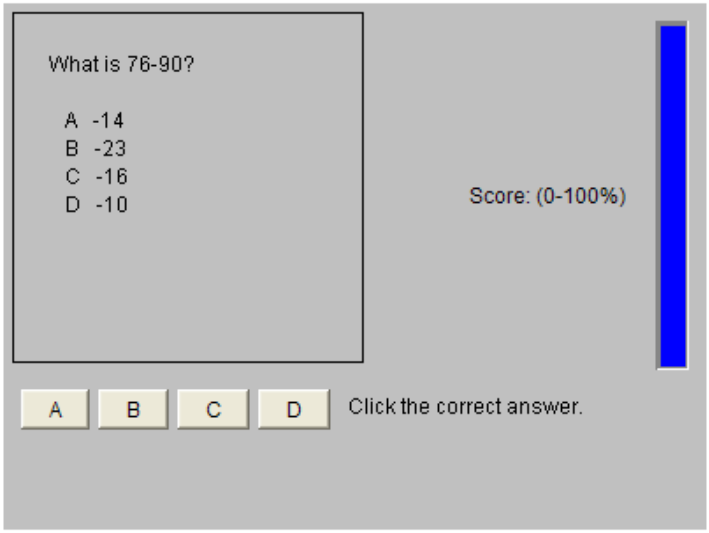


Figura 7

Las preguntas se presentan en forma aleatoria con el objeto de reducir la probabilidad de que una pregunta se formule 2 veces. Cuando se pulsa en una respuesta, la componente de software *Tcanv* se actualiza con nuevas preguntas e información del estado. El componente *Gauge* actualiza el resultado de las respuestas del usuario. El código se lista a continuación:

Quiz.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Quiz extends Applet {
 TCanv question = new TCanv();
 Gauge gauge = new Gauge();
 String labels[]={"   A   ","   B   ","   C   ","   D   "};
 Button button[] = new Button[labels.length];
 TCanv status=new TCanv();
 int questions = 0;
 int correctAnswers = 0;
 int currentAnswer;

 public void init() {
  Panel mainPanel = new Panel();
  Panel gaugePanel = new Panel();
  Panel bottomPanel = new Panel();
```

```
 Panel buttons = new Panel();
 question.setLeftMargin(20);
 question.setTopMargin(20);
 gauge.setHorizontal(false);
 gauge.setMaxValue(100.0);
 gauge.setCurrentValue(100.0);
 gauge.setHeight(200);
 gauge.setWidth(20);
 status.setHeight(20);
 status.setWidth(200);
 status.setTopMargin(0);
 status.setBorder(false);
 mainPanel.setLayout(new BorderLayout());
 mainPanel.add("Center",question);
 gaugePanel.add(new Label("Score: (0-100%)"));
 gaugePanel.add(gauge);
 mainPanel.add("East",gaugePanel);
 bottomPanel.setLayout(new BorderLayout());
 for(int i=0;i<labels.length;++i){
  button[i] = new Button(labels[i]);
  button[i].addActionListener(new ButtonHandler());
  buttons.add(button[i]);
 }
 buttons.add(status);
 bottomPanel.add("Center",buttons);
 mainPanel.add("South",bottomPanel);
 add(mainPanel);
}

public void start(){
 displayQuestion();
}

void displayQuestion() {
 question.setText(nextQuestion());
 if(questions==0) status.setText("Click the correct answer.");
 else{
  String s="Questions: "+String.valueOf(questions);
  s+=" Correct: "+String.valueOf(correctAnswers);
  status.setText(s);
 }
}

String nextQuestion() {
```

```
 String q = "What is ";
 String operand[] = {"+","-","*"};
 int op1 = randomInt(100);
 int op2 = randomInt(100);
 int op = randomInt(3);
 String operator = operand[op];
 int ans=0;
 switch(op){
 case 0:
  ans=op1+op2;
  break;
 case 1:
  ans=op1-op2;
  break;
 case 2:
  ans=op1*op2;
  break;
 }
 currentAnswer=randomInt(labels.length);
 q+=String.valueOf(op1)+operator+String.valueOf(op2)+"?| ";
 for(int i=0;i<labels.length;++i){
  q+="|"+labels[i];
  if(i==currentAnswer) q+=String.valueOf(ans);
  else{
   int delta = randomInt(10);
   if(delta==0) delta=1;
   int add = randomInt(2);
   if(add==1) q+=String.valueOf(ans+delta);
   else q+=String.valueOf(ans-delta);
  }
 }
 return q;
}

int randomInt(int max){
 int r = (int) (max*Math.random());
 r %= max;
 return r;
}

void answer(int i){
 ++questions;
 if(i==currentAnswer){
  ++correctAnswers;
```

```
    displayQuestion();
   }else{
    status.setText("Try again!");
   }
   double score = (double) correctAnswers/(double) questions;
   gauge.setCurrentValue(score*100.0);
   gauge.repaint();
   question.repaint();
   status.repaint();
  }

  class ButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent e){
    String s = e.getActionCommand();
    for(int i=0;i<labels.length;++i){
     if(labels[i].equals(s)){
      answer(i);
      break;
     }
    }
   }
  }
}
```

El applet *Quiz* es un ejemplo muy tosco del uso de los componentes de software en un applet. Normalmente, si estuviese utilizando componentes de software , armaría un applet por medio de una herramienta de programación visual. En este caso , puede ahorrarse casi toda la programación del applet.

El applet *Quiz* crea dos componentes *Tcanv* y los asignan a las variables *question* y *status*. Se crea un componente *Gauge* y se asigna a la variable *Gauge*. El componente que esta asignando a la variable *question* muestra el texto de una pregunta . El componente que esta asignando a la variable *status* muestra la información del estado a la derecha de los botones de respuesta .

El método *init()* del applet diseña el applet y establece las propiedades de los componentes de software. Los márgenes izquierdo y superior del componente de pregunta se establecen en 20 .El componente *Gauge* se cambia a vertical y su valor máximo se establece en 100.su valor activo también se establece en 100 , dando al usuario un voto de confianza. Las

dimensiones *width* y *height* del marcador también se modifican. Las dimensiones del componente *status* se ajustan . Su margen superior se establece en 0 y su *border* se desactiva.

El método *start()* del applet invoca sencillamente al método *displayQuestion()* con el fin de mostrar una pregunta al usuario .El método *displayQuestion()* invoca al método *setText()* del componente de la pregunta para mostrar el texto de dicha pregunta. El método *detText()* del componente *status* se invoca para mostrar información al usuario.

Las preguntas las crea el método *nextQuestion()*.Este método genera una pregunta aritmética en base a la suma , resta y multiplicación de los enteros que van entre 0 y 100 . Muestra la respuesta junto con otras 3 respuestas incorrectas :Estas respuestas aparecen en orden aleatorio.

El método *randomInt()* genera un entero aleatorio de 0 a 1 menos un máximo especificado.

El método *answer()* admite el manejo del los botones de respuesta comprobando si el usuario ha respondido correctamente y luego actualizando y mostrando *score*. Se invocan los métodos *repaint()* de los componentes de software para hacer que estos últimos actualicen sus visualizaciones .

La clase *ButtonHandler* admite el manejo de los eventos que están asociados con la pulsación de los botones de respuesta.

A continuación se muestra un archivo .html de manera de poder visualizar el applet creado.

quiz.htm

```
<HTML>
<HEAD>
<TITLE>Quiz</TITLE>
</HEAD>
<BODY>
<APPLET CODE="Quiz.class" WIDTH=400 HEIGHT=300>
[Quiz applet]
</APPLET>
</BODY>
```

Uso de la serialización (Quiz2.java)

En el applet *Quiz*, no hemos hecho uso de la persistencia. En lugar de que los componentes personalizables usen el BeanBox, el applet *Quiz* ha incluido un código especial en el método *init()* con el fin de realizar la edición y personalización de los componentes de software. El applet *Quiz2* que detallamos a continuación, no muestra como se usa la persistencia, pero si añade una característica que puede resultar interesante. En Quiz, la inicialización de las propiedades de los componentes de software, se realiza en el método *init()*. En Quiz2, los componentes *question*, *status* y *gauge* quedan personalizados en el BeanBox y escritos en los archivos *qcanv.ser*, *scan.ser* y *vgauge.ser*. Estos archivos no solo contienen información sobre las clases, sino también los valores de las propiedades personalizadas del componente de software. El método *instantiate()* de la clase Beans se usa para leer los componentes a partir de un almacenamiento serializado en los archivos .ser.

Quiz2.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class Quiz2 extends Applet {
// Declarar componentes de software
 TCanv question, status;
 Gauge gauge;
 String labels[]={"   A   ","   B   ","   C   ","   D   "};
 Button button[] = new Button[labels.length];
 int questions = 0;
 int correctAnswers = 0;
 int currentAnswer;

 public void init() {
  Panel mainPanel = new Panel();
  Panel gaugePanel = new Panel();
  Panel bottomPanel = new Panel();
```

```
 Panel buttons = new Panel();
 try{
// Cargar componentes serializados
  question = (TCanv) Beans.instantiate(null,"qcanv");
  gauge = (Gauge) Beans.instantiate(null,"vgauge");
  status = (TCanv) Beans.instantiate(null,"scanv");
 }catch(Exception ex){
  System.out.println("***"+ex.toString());
 }
 mainPanel.setLayout(new BorderLayout());
 mainPanel.add("Center",question);
 gaugePanel.add(new Label("Score: (0-100%)"));
 gaugePanel.add(gauge);
 mainPanel.add("East",gaugePanel);
 bottomPanel.setLayout(new BorderLayout());
 for(int i=0;i<labels.length;++i){
  button[i] = new Button(labels[i]);
  button[i].addActionListener(new ButtonHandler());
  buttons.add(button[i]);
 }
 buttons.add(status);
 bottomPanel.add("Center",buttons);
 mainPanel.add("South",bottomPanel);
 add(mainPanel);
}

public void start(){
 displayQuestion();
}

void displayQuestion() {
 question.setText(nextQuestion());
 if(questions==0) status.setText("Click the correct answer.");
 else{
  String s="Questions: "+String.valueOf(questions);
  s+=" Correct: "+String.valueOf(correctAnswers);
  status.setText(s);
 }
}

String nextQuestion() {
 String q = "What is ";
 String operand[] = {"+","-","*"};
 int op1 = randomInt(100);
```

```
 int op2 = randomInt(100);
 int op = randomInt(3);
 String operator = operand[op];
 int ans=0;
 switch(op){
 case 0:
  ans=op1+op2;
  break;
 case 1:
  ans=op1-op2;
  break;
 case 2:
  ans=op1*op2;
  break;
 }
 currentAnswer=randomInt(labels.length);
 q+=String.valueOf(op1)+operator+String.valueOf(op2)+"?| ";
 for(int i=0;i<labels.length;++i){
  q+="|"+labels[i];
  if(i==currentAnswer) q+=String.valueOf(ans);
  else{
   int delta = randomInt(10);
   if(delta==0) delta=1;
   int add = randomInt(2);
   if(add==1) q+=String.valueOf(ans+delta);
   else q+=String.valueOf(ans-delta);
  }
 }
 return q;
}

int randomInt(int max){
 int r = (int) (max*Math.random());
 r %= max;
 return r;
}

void answer(int i){
 ++questions;
 if(i==currentAnswer){
  ++correctAnswers;
  displayQuestion();
 }else{
```

```
    status.setText("Try again!");
   }
// Actualizar propiedades del componente
   double score = (double) correctAnswers/(double) questions;
   gauge.setCurrentValue(score*100.0);
   gauge.repaint();
   question.repaint();
   status.repaint();
  }

  class ButtonHandler implements ActionListener {
   public void actionPerformed(ActionEvent e){
    String s = e.getActionCommand();
    for(int i=0;i<labels.length;++i){
     if(labels[i].equals(s)){
      answer(i);
      break;
     }
    }
   }
  }
}
```

Mostramos el archivo *quiz2.htm* que inserta el applet creado en una pagina Web.

quiz2.htm

```
<HTML>
<HEAD>
<TITLE>Quiz</TITLE>
</HEAD>
<BODY>
<APPLET CODE="Quiz2.class" WIDTH=400 HEIGHT=300>
[Quiz applet]
</APPLET>
</BODY>
</HTML>
```

Creación de los archivos *.ser*

Hemos utilizado el BeanBox para personalizar cada uno de los componentes de software utilizado por Quiz2 y los hemos guardado en archivos .ser por medio de la orden SerializeComponent del menú BeanBox File como se muestra en la Figura 8.

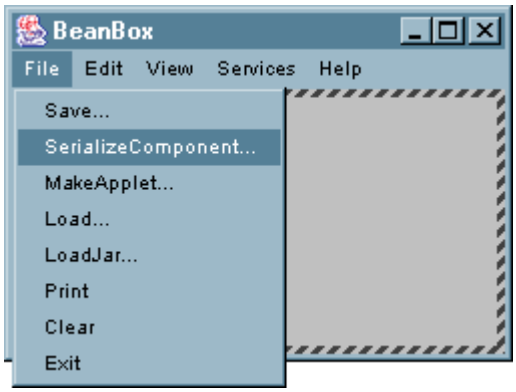


Figura 8

De esta manera, podemos cambiar las propiedades de los componentes de software con la orden SerializeComponent descrita anteriormente (guardando estos cambios en un archivo .ser) y permitir así que estos cambios se vean reflejados al momento de ejecutar nuevamente nuestro programa.

Conclusiones

Al concluir este proyecto, podemos apreciar la utilidad que presenta el uso de Java Beans. Con esta herramienta resulta muy sencillo escribir programas complejos utilizando pequeños componentes de software que pueden ser unidos a través de la herramienta de diseño visual (gratuita) BDK 1.1 u otras herramientas de diseño visual.

En este estudio descubrimos todas las ventajas que presentan los Java Beans por medio de los ejemplos presentados en los anexos del trabajo así como también en el análisis del paquete Java.beans.

REFERENCIAS:

Java 1.2 al descubierto ,Jaime Jaworski.

Descubre Jav 1.2 (Prentice Hall).

http://www.programacion.com/java/tutorial/beans/

http://profesores.elo.utfsm.cl/%7Eagv/elo330/2s03/index.html

http://scsx01.sc.ehu.es/sbweb/fisica/cursoJava/applets/javaBeans/fundamento.htm