UD3. Creación de compoñentes visuais

RA3. Crea compoñentes visuais, para o que valora e emprega ferramentas específicas.

- CA3.1. Identificáronse as ferramentas para deseño e proba de compoñentes.
- CA3.2. Creáronse compoñentes visuais.
- CA3.3. Definíronse as súas propiedades e asignáronse valores por defecto.
- CA3.4. Determináronse os eventos aos que debe responder o compoñente e asociáronselles as accións correspondentes.
- CA3.5. Realizáronse probas unitarias sobre os compoñentes desenvolvidos.
- CA3.6. Documentáronse os compoñentes creados.
- CA3.7. Empaquetáronse compoñentes.
- CA3.8. Programáronse aplicacións cuxa interface gráfica utilice os compoñentes creados.

Creación de compoñentes visuais

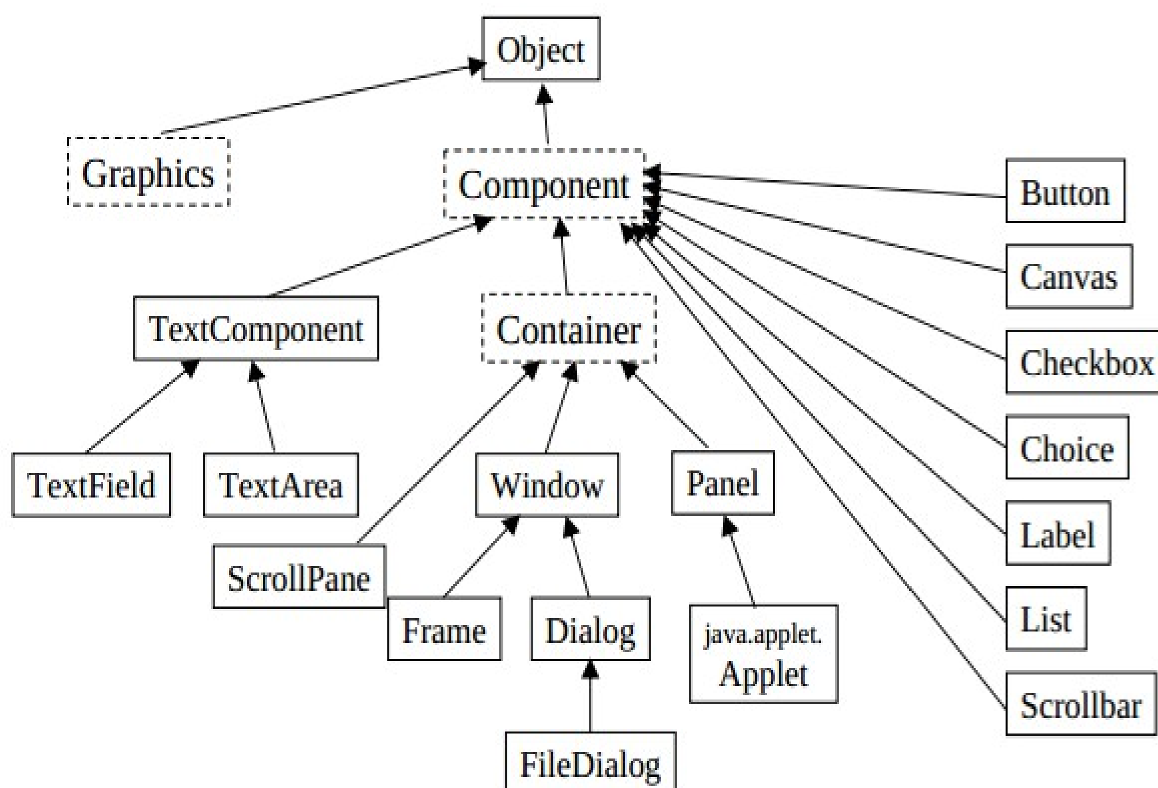
- Concepto e características dos compoñentes.
- Propiedades e atributos.
- Eventos: asociación de accións a eventos.
- Persistencia do compoñente.
- Ferramentas para desenvolvemento de compoñentes visuais.
- Empaquetaxe de compoñentes.

1. Componentes visuales de una interfaz gráfica.

Una aplicación gráfica puede tener **componentes visuales** o que vemos (botones, cajas de texto...) o **componentes no-visuales** que no están a la vista del usuario (temporizadores, conexiones a bases de datos...)

Los **componentes GUI (widgets en lo sucesivo)** de una interfaz son los objetos visuales de la misma. Un programa gráfico es un conjunto de **componentes anidados** (ventanas, contenedores, menús, barras, botones, campos de texto, etc...)

Un aspecto importante es la **disposición (layout)** es decir, cómo se colocan los componentes para lograr un GUI cómodo de utilizar. Los **layout managers** gestionan la organización de los componentes gráficos de la interfaz.



Esquema de componentes gráficos de IU de Java

Los componentes visuales podemos dividirlos a su vez en dos tipos:

- **Componentes interactivos**: permiten que el usuario final los manipule, ya sea introduciendo datos, seleccionado elementos, etc. De forma que estos componentes pueden recibir el foco, es decir, permite enviar el cursor a cualquier zona de la interfaz (con setFocus) así como los eventos propios del teclado y del ratón. Normalmente, el **propio sistema operativo es el encargado de dibujar** el aspecto del componente, haciendo el componente las llamadas correspondientes para que este aspecto cambie.

- **Componentes gráficos**: el propio componente es el encargado de dibujar en la pantalla lo que crea oportuno, bien a través de las funciones básicas del API de Windows según el IDE utilizado o bien a través de otras librerías gráficas, como OpenGL, DirectX, etc. Estos componentes, no suelen recibir eventos del usuario final, aunque si eventos del propio programador, ya que su cometido no suele ir más allá de mostrar ciertos gráficos o imágenes en la pantalla.

Habiendo muchos componentes si tuviésemos que elegir los más habituales (hasta 15) en una ventana o GUI estos serían quizás los más importantes:

- window
- button
- label
- contenedores
- checkbox
- textarea
- list
- dialog
- scroolbar
- textcomponent
- window
- panel
- combobox
- textbox o caja de texto
- menubar...

Aunque hay bastantes más.

Como comentamos antes la distribución de los componentes en la ventana es fundamental para la **legibilidad de la misma** por el usuario.

Para agrupar/clasificar/distinguir elementos dentro de una ventana, se suele dividir ésta en **áreas.**

Las áreas se pueden formar:

– Utilizando **elementos visuales distintivos**, como líneas de división, tipos de letra, colores, etc.

– Las etiquetas **para titular una sección**, un grupo de elementos o un elemento simple pueden ser independientes (Label) o el dato de título (Caption) de un objeto gráfico.

– Utilizando **objetos contenedores**, que usualmente imponen algun tipo de organización y que suelen ser lo más recomendable porque facilita la manipulación del diseño de la

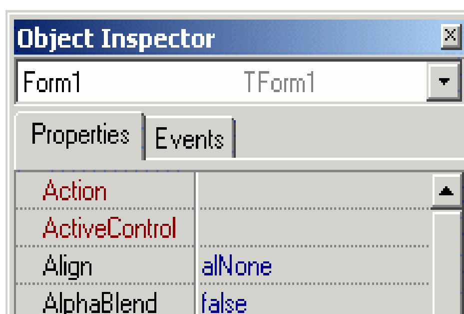
interfaz.

2. Propiedades e atributos de los componentes visuales

Como hemos dicho varias veces a lo largo del curso los **componentes visuales**, a diferencia de los no visuales, son aquellos que, al utilizarlos, muestran algún elemento (o dibujo) en la pantalla y es el usuario de nuestros programas el que interactúa con él. El componente es el principal responsable de dibujar en la pantalla lo que sea oportuno, dependiendo de su estado, del valor de sus atributos, etc. Hay muchos componentes de este tipo, como pueden ser los botones (tButton), etiquetas de texto (tLabel), formas (tShape), etc.

Todo componente tiene **propiedades, eventos, métodos y atributos**

- Las **propiedades** son datos públicos del componente, muy **parecidas a los atributos de una clase**, aunque se accede a ellas a través de dos métodos: un método para leer su valor, y otro para modificarlo. Existen propiedades de sólo lectura, en las que podemos consultar pero no modificar su valor, y propiedades de sólo escritura. Por ejemplo, las propiedades "Alto" (Width) y "Ancho" (Height) de un botón permiten que un programador pueda cambiar las dimensiones del componente. Cuando el programador cambia alguna de ellas, el componente debe redibujarse en la pantalla, para mostrar los nuevos cambios.



- Los **atributos** . Tienen la misma misión que en programación orientada a objetos, es decir: almacenar datos internos al objeto (o clase). En el mundo de los componentes, los atributos siempre son internos y de uso privado, y debemos utilizar las propiedades para que un programador pueda leer o establecer un dato. Por ejemplo, propiedad ***color*** le corresponde ***atributos*** *rojo, verde....*

No hay que confundir "propiedad" con "atributo" :

1. Ambos describen el estado interno y comportamiento, pero en dos contextos diferentes
2. Normalmente los atributos nunca son públicos. En cambio las propiedades siempre son características públicas de los componentes
3. Los atributos son accesibles únicamente a nivel de programación. Las propiedades son

accesibles tanto a nivel de programación como interactivamente desde el entorno de desarrollo

- Finalmente los **eventos** los que le dan funcionalidad a los **widgets** y de los que hablaremos a continuación. Los eventos viene regidos por los **métodos o funciones** que son llamados desde el programa por acción del usuario. Cada tipo de objeto o control tiene sus propios métodos.

3. Eventos: asociación de accións a eventos.

Este cuestiones ya las analizamos en unidades didácticas anteriores pero haremos hincapié y profundizaremos en el tema.

Un **evento** es una **acción realizada por el usuario de una interfaz** y que da lugar a un efecto mediante la ejecución de una función.

El **ratón**, por ejemplo y según el lenguaje de programación utilizado, tiene los siguientes eventos:

- *Mouse Over*. Se produce cuando el puntero del ratón se encuentra por encima de una determinada zona.
- *Mouse Out*. Se produce cuando el puntero del ratón abandona una determinada zona.
- *Mouse Clicked*. Se produce cuando se pulsa un botón del ratón.
- *Mouse Double-Clicked*. Se produce cuando se pulsa dos veces en un intervalo pequeño de tiempo un botón del ratón.

También depende del objeto sobre el que actúa, si es un enlace directo lanza una aplicación si es sobre un CheckBox, activa y desactiva.

Por otro lado los eventos vinculados al **teclado** y también dependiendo del tipo de lenguaje y objeto sobre el que actúa puede ser:

- *onKeyPress:* evento que se activa cuando un usuario pulsa y deja de pulsar una tecla o también cuando la mantiene pulsada;o
- *onKeyDown*: activado cuando se pulsa la tecla;
- *onKeyUp*: activado cuando una tecla, que se había pulsado, deja de pulsarse;

Similar al caso anterior puede haber más eventos pero estos son los más utilizados.

La **programación dirigida por eventos** es un paradigma de la programación el que tanto la estructura como la **ejecución de los programas** van determinados por los **sucesos** que ocurran en el sistema, definidos por el **usuario o que ellos mismos provoquen**.

Para entender la programación dirigida por eventos, podemos oponerla a lo que no es: mientras en la programación secuencial o estructurada es el programador el que define cuál va a ser el flujo del programa, en la programación dirigida por eventos **será el propio usuario** —o lo que sea que esté accionando el programa— el que dirija el flujo del programa. Aunque en la programación secuencial puede haber intervención de un agente externo al programa, estas intervenciones ocurrirán cuando el programador lo haya determinado, y no en cualquier momento como puede ser en el caso de la programación dirigida por eventos.

El creador de un programa dirigido por eventos **debe definir los eventos** que manejarán su programa y las acciones que se realizarán al producirse cada uno de ellos, lo que se conoce como el administrador de eventos. Los eventos soportados estarán determinados por el lenguaje de programación soportado por el SO o los creados por el programador.

En la programación dirigida por eventos, al comenzar la ejecución del programa se llevarán a cabo las inicializaciones y demás código inicial y a continuación el programa quedará **bloqueado hasta que se produzca algún evento**. Cuando alguno de los eventos esperados por el programa tenga lugar, el programa pasará a ejecutar el código del correspondiente al administrador o **manejador del evento**. Por ejemplo, si el evento consiste en que el usuario ha hecho click en el botón de play de un reproductor de películas, se ejecutará el código del manejador que será el que haga que la película se muestre por pantalla.

Un ejemplo claro lo tenemos en los sistemas de programación del IDE **Visual Basic**, en los que a cada **elemento del programa** (objetos, controles, etcétera) **se le asignan una serie de eventos** que generará dicho elemento, como la pulsación de un botón del ratón sobre él o el redibujado del control.

La programación dirigida por eventos **es la base** de lo que llamamos **interfaz de usuario**, aunque puede emplearse también para desarrollar interfaces entre componentes de Software.

La programación orientada a eventos permite **interactuar con el usuario en cualquier momento de la ejecución.** Esto se consigue debido a que los programas creados bajo esta arquitectura se componen por un bucle exterior permanente encargado de recoger los eventos, y distintos procesos que se encargan de tratarlos. Habitualmente, este bucle externo permanece oculto al programador que simplemente se encarga de tratar los eventos, aunque en algunos lenguajes será necesaria su construcción.

```
While (true){
    Switch (event){
        case mousse_button_down:
        case mouse_click:
        case keypressed:
        case Else:
    }
}
```

La programación orientada a eventos supone una complicación añadida con respecto a otros tipos de programación, debido a que el **flujo de ejecución del software escapa al control del programador.** En cierta manera podríamos decir que en la programación clásica el flujo estaba en poder del programador y era este quién decidía el orden de ejecución de los procesos, mientras que en programación orientada a eventos, es el usuario el que controla el flujo y decide.

Pongamos como ejemplo de la problemática existente: un menú con dos botones, *botón 1 y botón 2*. Cuando el usuario pulsa *botón 1,* el programa se encarga de recoger ciertos parámetros que están almacenados en un fichero y calcular algunas variables. Cuando el usuario pulsa el *botón 2,* se le muestran al usuario por pantalla dichas variables. Es sencillo darse cuenta de que la naturaleza indeterminada de las acciones del usuario y las características de este paradigma pueden fácilmente desembocar en el error fatal de que se pulse el *botón 2* sin previamente haber sido pulsado el *botón 1.* Aunque esto no pasa si se tienen en cuenta las propiedades de dichos botones, haciendo inaccesible la pulsación sobre el *botón 2* hasta que previamente se haya pulsado el *botón 1.*

Con la evolución de los lenguajes orientados a eventos, la interacción del software con el usuario ha mejorado enormemente permitiendo la aparición de interfaces que, aparte de ser la vía de comunicación del programa con el usuario, son la propia apariencia del mismo. Estas interfaces, también llamadas **GUI (Graphical User Interface)**, han sido la herramienta imprescindible para acercar la informática a los usuarios, permitiendo en muchos casos, a principiantes utilizar de manera intuitiva y sin necesidad de grandes conocimientos, el software que ha colaborado a mejorar la productividad en muchas tareas.

Con todo ello observamos que en la **programación orientada a eventos** no existe un único flujo de ejecución y que el funcionamiento es **asíncrono** porque depende de acciones externas al mismo. Aún así los tipos de eventos los podemos dividir en:

- **externos:** producidos por el usuario (teclado o ratón)
- **internos:** producidos por el sistema (temporizadores, transmisiones de datos..)

Uno de los periféricos que ha cobrado mayor importancia tras la aparición de los programas orientados a eventos ha sido el ratón, gracias también en parte a la aparición de los sistemas operativos modernos con sus interfaces gráficas. Sus interfaces graficas suelen dirigir directamente al controlador interior que va entrelazado al algoritmo.

4. Persistencia del componente

La persistencia permite al programador **almacenar, transferir y recuperar** el estado de los objetos. Para esto existen varias técnicas:

- **serialización** (o *marshalling* en inglés) consiste en un proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML, entre otros. La serie de bytes o el formato pueden ser usados para crear un nuevo objeto que es idéntico en todo al original, incluido su estado interno (por tanto, el nuevo objeto es un clon del original). La **serialización** es un mecanismo ampliamente usado para transportar objetos a través de una red, para hacer persistente un objeto en un archivo o base de datos, o para distribuir objetos idénticos a varias aplicaciones o localizaciones.

  La **serialización** tiene una serie de ventajas:

  - Un método de persistencia de objetos que es más conveniente que escribir sus propiedades a un archivo de texto en disco.
  - Un método de emisión de llamadas a procedimiento remoto, por ejemplo, como en SOAP.
  - Un método para la distribución de objetos, especialmente en los componentes software, tales como COM, CORBA, etc.
  - Un método para detectar cambios en variables en el tiempo.

- **base de datos orientada a objetos**, la información se representa mediante objetos como los presentes en la POO. Un **sistema gestor de base de datos orientada a objetos** (**ODBMS**, *object database management system*) hace que los objetos de la base de datos aparezcan como objetos de un lenguaje de programación en uno o más lenguajes de programación a los que dé soporte. Un ODBMS extiende los lenguajes con datos persistentes de forma transparente, control de concurrencia, recuperación de datos, consultas asociativas y otras capacidades.

  Las bases de datos orientadas a objetos se diseñan para trabajar bien en conjunción con lenguajes de programación orientados a objetos como Java, C#, Visual Basic.NET y C++. Los ODBMS son una buena elección para aquellos sistemas que necesitan un buen rendimiento en la manipulación de tipos de dato complejos. Los ODBMS proporcionan los costes de desarrollo más bajos y el mejor rendimiento cuando se usan objetos gracias a que almacenan objetos en disco y tienen una integración transparente con el programa

escrito en un lenguaje de programación orientado a objetos, al almacenar exactamente el modelo de objeto usado a nivel aplicativo, lo que reduce los costes de desarrollo y mantenimiento.

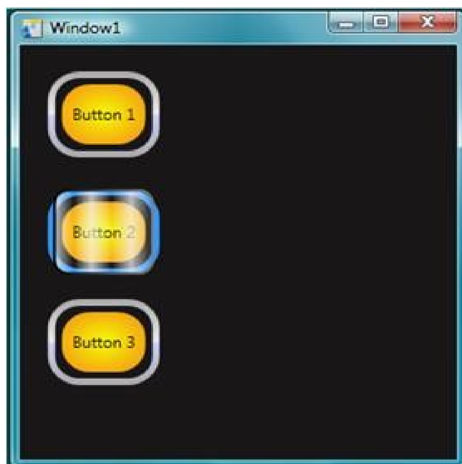
En años recientes, han aparecido muchos prototipos experimentales y sistemas de bases de datos comerciales orientados a objetos. Entre los primeros se encuentran los sistemas **ORION**, **OpenOODB**, **IRIS**, **ODE** y el proyecto **ENCORE/ObServer**. Y entre los sistemas disponibles en el mercado están: **GEMSTONE/OPAL** de ServicLogic, **ONTOS** de Ontologic, **Objectivity** de Objectivity Inc.,**Versant** de Versant Technologies, **ObjecStore** de Object Design y **O2** de O2 Technology.

- **motor de persistencia** encargado de traducir entre los dos formatos de datos: **de registros a objetos y de objetos a registros.** Cuando el programa quiere grabar un objeto llama al motor de persistencia, que traduce el objeto a registros y llama a la base de datos para que guarde estos registros. De la misma manera, cuando el programa quiere recuperar un objeto, la base de datos recupera los registros correspondientes, los cuales son traducidos en formato de objeto por el motor de persistencia. Una ventaja del motor de persistencia es que es el mismo para todas las aplicaciones. De esta forma sólo debe programarse una vez y puede usarse para todas las aplicaciones que se desarrollen.

5. Ferramentas para desenvolvemento de compoñentes visuais.

Con cualquier herramienta de diseño gráfico podemos diseñar diferentes tipos de componentes visuales, preferentemente botones e iconos ... Una herramienta libre y muy vérsatil es **Gimp.** A la hora de diseñar un componente visual es importante guardar la una relación adecuada en sus dimensiones.

Otro ejemplo es **Microsoft Expression Blend** es una herramienta profesional desarrollado por Microsoft, de diseño que permite controlar la eficacia del XAML, .NET y Silverlight para proporcionar experiencias de usuario atractivas como la generación de nuevos botones, cajas de texto... en escritorios conectados y Web.



Diseño de Botones con Expression Blend

6. Empaquetamiento de componentes.

Una vez diseñados los componentes queda incluirlos en el entorno de desarrollo o en la aplicación. En las prácticas plantearemos la modificación del diseño de algunos componentes.