Componentes

¿Qué son los componentes?

Se extiende el concepto de objeto de la programación orientada a objetos:

> Un componente es ***un objeto*** **que tiene una representación visual y puede interactuar** con el usuario y con otros componentes.

Además, puede estar **compuesto por otros componentes**, que constituyen el pilar de construcción de las interfaces gráficas modernas.

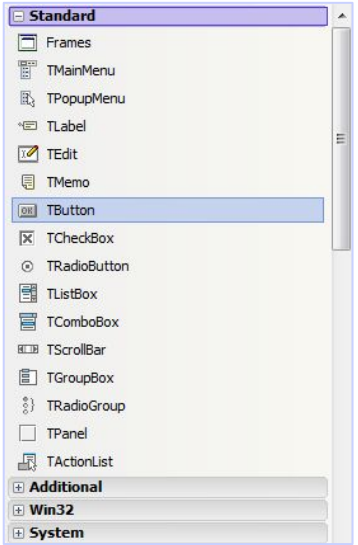
Cuando pensamos en **botones** o **campos de texto** estamos considerando componentes con los que, a su vez, podemos generar otros componentes **definidos por el usuario.**

Historia de la Programación Orientada a Componentes

Surgimiento de los componentes (Décadas de los´70 y 80)

Aparecieron en **lenguajes *Object-Oriented*** como Smalltalk, Visual Basic y Delphi.

- Nacieron con la idea de **reutilización** de *software.*
- Se popularizaron para **definir interfaces** gráficas.
- Se integraron con **paletas de componentes** en las primeras IDEs.
- Se utilizaron para generar **aplicaciones de escritorio.**
- Se enfocaron en la **encapsulación** de funcionalidad.
- Propusieron una presentación de **interfaces estándar.**



Evolución y estándares (Década de los´90)

- Aparecieron estándares como COM/DCOM (Microsoft).
  - Buscaban **independencia del lenguaje** de programación.
  - Permitían **reutilizar componentes** complejos en distintas aplicaciones.
  - Establecían estándares para la **interoperabilidad y la integración** en aplicaciones empresariales.
- En el ecosistema Java, **aparecieron los `JavaBeans`** (clase Java que cumplía con ciertas convenciones de diseño y se utilizaba, principalmente, para encapsular datos en aplicaciones Java).

Estos estándares fueron perdiendo relevancia con la evolución de nuevas tecnologías.

Expansión en la web (Década de los ´2000)

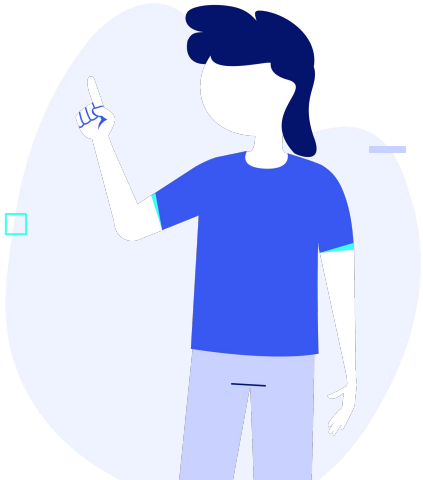
- **Se transicionó hacia el desarrollo web y móvil**: Se dejó atrás el modelo *desktop* predominante.
- **Se popularizó JavaScript**: Como principal lenguaje para el desarrollo web.
- **Surgieron las librerías JS**: Como jQuery que simplificó la manipulación del DOM pero sin un enfoque declarativo ni una estructura componentizada.
- **Complejidad en interfaces gráficas**: Cada vez más desafiantes y difíciles de mantener.

En este contexto, la **necesidad de componentes reutilizables** se hizo evidente. Esto preparó el terreno para la evolución hacia los *frameworks* modernos, que ofrecieran soluciones más estructuradas y componentizadas, en las décadas siguientes.

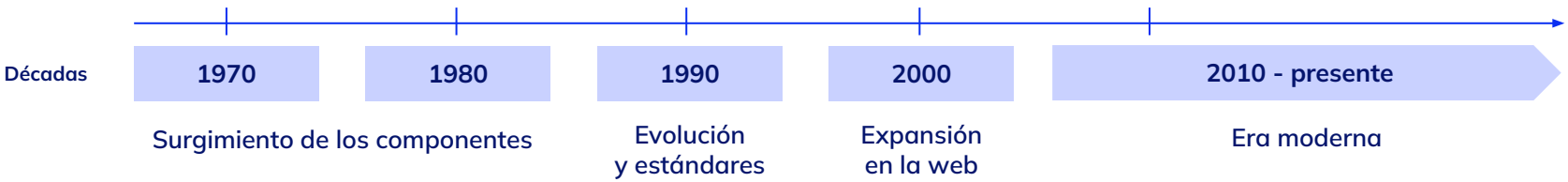
Era moderna (Década de 2010 - Presente)

- **Surgieron *frameworks* basados en componentes:** como React, Angular, Vue.js, Blazor y Flutter.
- **Se fomentaron las aplicaciones móviles** y el desarrollo multiplataforma.
- **Enfoque modular** para sistemas escalables y mantenibles con componentes independientes de interfaz de usuario.
- **Se adoptaron estándares para *Web Components:*** Se fortaleció la reutilización de componentes a través de diferentes tecnologías web.

La **integración generalizada de componentes** en *frameworks* modernos y tecnologías como *Web Components* ha revolucionado el desarrollo de *software*. Esto permitió la **creación de aplicaciones más escalables y modulares en diversos entornos digitales.**



Resumen de la historia de la Programación Orientada a Componentes



Beneficios de utilizar componentes

- **Reutilización:** Los componentes permiten reutilizar código y funcionalidades en múltiples partes de una aplicación, reducen la duplicación de esfuerzos y facilitan el mantenimiento.
- **Modularidad:** La estructura basada en componentes facilita la división del desarrollo en partes más pequeñas y manejables, lo que simplifica la colaboración entre equipos y mejora la escalabilidad del proyecto.
- **Consistencia:** Al utilizar componentes, se asegura la coherencia visual y funcional a lo largo de toda la aplicación. Esto proporciona una experiencia de usuario más uniforme.
- **Facilidad en la mantenibilidad:** Los cambios en un componente pueden aplicarse de manera centralizada y propagarse automáticamente a todas las instancias donde se utiliza. Esto simplifica la actualización y la corrección de errores.

- **Desarrollo más rápido:** La capacidad de reutilizar componentes existentes acelera el proceso de desarrollo. Esto permite que los equipos se enfoquen en la implementación de nuevas funcionalidades.
- **Compatibilidad multiplataforma:** Los componentes son diseñados para ser independientes de la plataforma. Esto facilita su uso en aplicaciones que se despliegan en diferentes sistemas operativos y dispositivos, reduce la complejidad del desarrollo multiplataforma.

educación IT
CODE YOUR FUTURE

Componentes en JavaScript

Definición de componente en JavaScript “Vainilla”

- **Los componentes son elementos autónomos,** representan partes independientes y reutilizables de la interfaz de usuario.
- **Poseen estructura y comportamiento encapsulados.** Cada componente contiene su propio HTML, CSS y lógica JavaScript.
- **Se utilizan clases** para definir componentes con propiedades y métodos encapsulados.

Ejemplo:

```
class Componente { ... }
```

- **Funciones en JavaScript:** Definición de funciones que devuelven elementos HTML con comportamiento específico.

Ejemplo:

```
function render() { ... return elemento; }
```

Vainilla significa: JavaScript estándar sin uso de librerías.

Así se vería un **componente botón** desarrollado en JavaScript:

```
class Boton {
    constructor(nombre, onClick, parent) {
        this.nombre = nombre;
        this.onClick = onClick;
        this.parent = parent;
    }

    render() {
        const button = document.createElement('button');
        button.textContent = this.nombre;
        button.addEventListener('click', this.onClick);
        this.parent.appendChild(button);
    }
}
```

Eventos

Representan **acciones o sucesos específicos** que ocurren **dentro de un componente** o interfaz de usuario como clics de botón, cambios de estado, ingreso de datos, entre otros.

Los eventos permiten:

- **Extender el comportamiento** del componente, en el contexto de una aplicación.
- Realizar **composición de componentes.**

> En el ejemplo anterior, la **propiedad `onClick`** del componente permite adaptar su comportamiento cuando se interactúa con él.

Captura y manejo:

Los eventos son capturados y manejados mediante funciones o métodos específicos dentro de los componentes.

Asociación de eventos:

Se asocian funciones o métodos a eventos particulares para definir la respuesta del componente ante acciones del usuario.