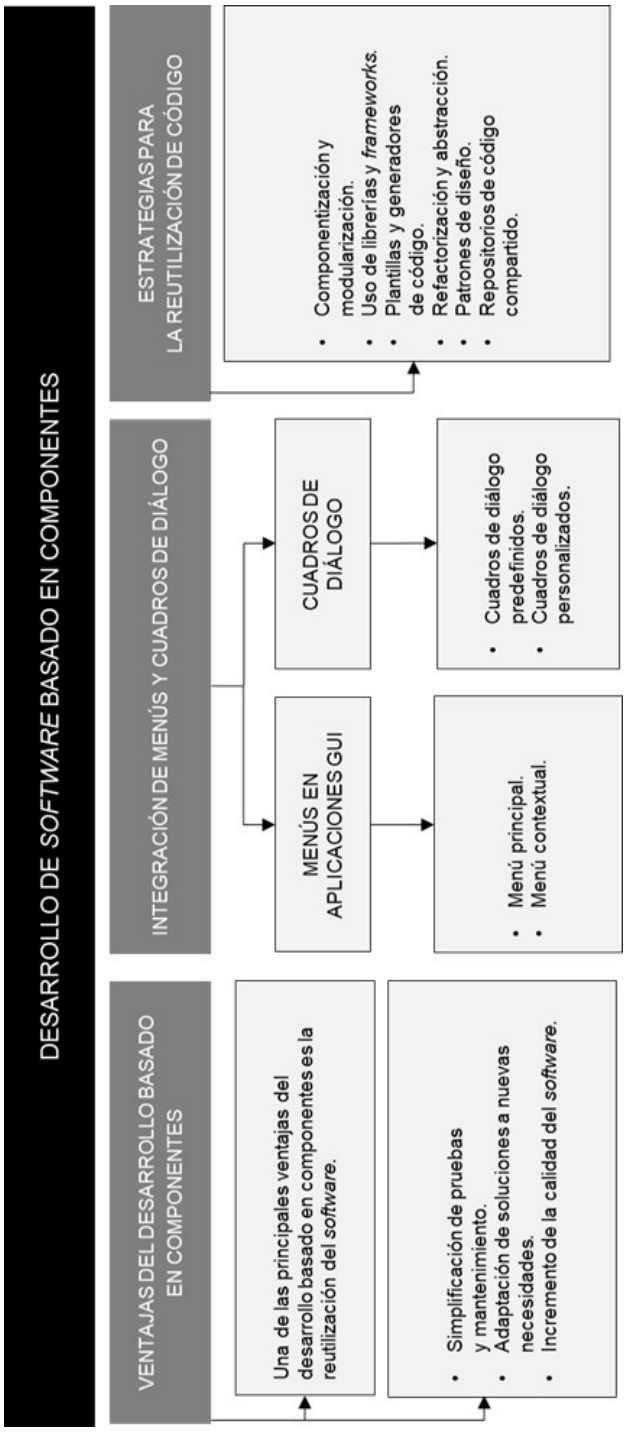
Desarrollo de Interfaces

Tema 3. Desarrollo de software basado en componentes

Índice


DESARROLLO DE *SOFTWARE* BASADO EN COMPONENTES
VENTAJAS DEL DESARROLLO BASADO EN COMPONENTES
Una de las principales ventajas del desarrollo basado en componentes es la reutilización del *software*.
Simplificación de pruebas y mantenimiento.
Adaptación de soluciones a nuevas necesidades.
Incremento de la calidad del *software*.
INTEGRACIÓN DE MENÚS Y CUADROS DE DIÁLOGO
MENÚS EN APLICACIONES GUI
Menú principal.
Menú contextual.
CUADROS DE DIÁLOGO
Cuadros de diálogo predefinidos.
Cuadros de diálogo personalizados.
ESTRATEGIAS PARA LA REUTILIZACIÓN DE CÓDIGO
Componentización y modularización.
Uso de librerías y *frameworks*.
Plantillas y generadores de código.
Refactorización y abstracción.
Patrones de diseño.
Repositorios de código compartido.

3.1. Introducción y objetivos

El desarrollo de *software* basado en componentes se ha consolidado como una metodología esencial en la ingeniería de *software* moderna, ofreciendo múltiples ventajas que permiten a los desarrolladores crear aplicaciones más eficientes, escalables y mantenibles. Este enfoque se centra en la construcción de aplicaciones mediante la combinación de **componentes modulares,** que encapsulan funcionalidades específicas y que pueden ser reutilizados en diferentes partes del proyecto o incluso en proyectos completamente distintos. Esta modularidad no solo simplifica el desarrollo, sino que también promueve la **reutilización de código,** un concepto clave para mejorar la productividad y la calidad del *software.*

Uno de los principales beneficios del desarrollo basado en componentes es la capacidad de **reutilización del código.** Al crear componentes modulares e independientes, los desarrolladores pueden evitar la duplicación de esfuerzos y asegurarse de que las soluciones comprobadas sean aplicadas consistentemente en toda la aplicación. Esta reutilización se ve facilitada por estrategias específicas, como l a **componentización,** el uso de **librerías** y ***frameworks,*** y la aplicación de **patrones de diseño.** Cada una de estas estrategias contribuye a la creación de un código que no solo es más fácil de mantener, sino que también puede adaptarse más rápidamente a los cambios en los requisitos del proyecto.

Además, la integración efectiva de estos componentes en el entorno de desarrollo es fundamental. Esto implica comprender y gestionar las propiedades de los componentes para garantizar que interactúen de manera coherente y eficiente dentro de la aplicación. La correcta gestión de estas propiedades facilita la creación de interfaces de usuario ricas y funcionales, donde cada componente puede ser ajustado y personalizado según las necesidades específicas del proyecto.

Por último, las estrategias para la reutilización de código van más allá de la simple reutilización de componentes. Incluyen prácticas como la **refactorización,** que mejora la estructura y legibilidad del código, y la **abstracción,** que permite crear interfaces generales para manejar comportamientos comunes. Estas prácticas, combinadas con el uso de plantillas, generadores de código y repositorios de código compartidos, aseguran que los desarrolladores puedan crear soluciones flexibles y escalables que se puedan adaptar a las futuras necesidades del proyecto.

Los **objetivos** específicos que se pretenden alcanzar con este tema son:

- **Comprender los beneficios del desarrollo basado en componentes:** adquirir una comprensión profunda de cómo la modularización y la componentización pueden mejorar la eficiencia y la calidad del *software.*
- **Aplicar estrategias efectivas para la reutilización de código:** desarrollar habilidades para identificar y aplicar las mejores prácticas para reutilizar código de manera efectiva en diferentes contextos.
- **Gestionar propiedades de componentes en entornos de desarrollo:** aprender a manejar las propiedades de los componentes para garantizar una integración coherente y eficiente en la aplicación.
- **Implementar patrones de diseño y refactorización:** desarrollar la capacidad de aplicar patrones de diseño y técnicas de refactorización para mejorar la mantenibilidad y escalabilidad del código.

3.2. Ventajas del desarrollo basado en componentes

El desarrollo de *software* basado en componentes (CBD, por sus siglas en inglés) ha surgido como una metodología predominante en la ingeniería de *software,* con múltiples ventajas que lo hacen atractivo tanto para desarrolladores como para empresas. Este enfoque se centra en la construcción de aplicaciones mediante la **combinación** de **componentes** de *software* **reutilizables,** lo que permite una serie de beneficios significativos en el proceso de desarrollo.

Una de las principales **ventajas** del desarrollo basado en componentes es la **reutilización del *software.*** Al construir aplicaciones utilizando componentes ya existentes, se evita la necesidad de desarrollar funcionalidades desde cero. Esto no solo reduce los costos y el tiempo de desarrollo, sino que también incrementa la fiabilidad del *software.* Los componentes reutilizables, al haber sido probados en múltiples aplicaciones, suelen ser más estables y menos propensos a errores. Además, este enfoque facilita la adaptación de soluciones a nuevas necesidades o proyectos, lo que permite a las organizaciones responder con mayor agilidad a las demandas del mercado.

Otra ventaja clave es la **simplificación** en las **pruebas** y el **mantenimiento.** Dado que los componentes son unidades modulares e independientes, es posible probarlos de manera aislada antes de integrarlos en el sistema completo. Esto reduce la complejidad de las pruebas, ya que los desarrolladores pueden enfocarse en verificar la funcionalidad de cada componente por separado. Una vez integrados, las pruebas a nivel de sistema se simplifican debido a la estabilidad inherente de los componentes ya probados. Además, el mantenimiento del *software* se vuelve más manejable, ya que las actualizaciones o correcciones pueden realizarse en componentes específicos sin afectar al resto del sistema.

El desarrollo basado en componentes también **incrementa** la **calidad** del ***software.*** Al utilizar componentes que han sido optimizados y mejorados a lo largo del tiempo, se asegura un mayor nivel de calidad en el producto final. Estos componentes son frecuentemente actualizados y refinados por expertos, lo que contribuye a la creación de aplicaciones robustas y confiables. Además, el uso de componentes estandarizados facilita la **interoperabilidad** entre diferentes sistemas y plataformas, ampliando la versatilidad del *software* desarrollado.

Una ventaja adicional es la **reducción del riesgo** en proyectos de desarrollo de *software.* El CBD permite identificar y mitigar riesgos potenciales al aprovechar componentes que han sido previamente probados en otros entornos. Esto es, especialmente, útil en proyectos complejos, donde la incertidumbre puede ser alta y el margen de error, reducido. Al utilizar componentes confiables, se minimizan las sorpresas durante las fases críticas de desarrollo e implementación.

Por último, el desarrollo basado en componentes promueve la **integración continua** y la **mejora incremental del *software.*** En lugar de desarrollar una solución completa desde cero, los equipos pueden construir y mejorar las aplicaciones de manera iterativa, integrando nuevos componentes a medida que estén disponibles o se desarrollen nuevas funcionalidades. Esto no solo mejora la eficiencia del desarrollo, sino que también permite a las empresas mantener sus aplicaciones actualizadas y alineadas con las últimas innovaciones tecnológicas.

En resumen, el desarrollo basado en componentes ofrece ventajas significativas que van desde la reutilización y la mejora de la calidad del *software* hasta la reducción de riesgos y la simplificación del mantenimiento. Estas características hacen que el CBD sea una opción preferida para el desarrollo de aplicaciones modernas, donde la agilidad, la fiabilidad y la eficiencia son esenciales para el éxito.

Componente modal reutilizable

El siguiente ejemplo muestra un componente modal que puede ser reutilizado en diferentes partes de la aplicación para mostrar mensajes o formularios.

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Componente Modal</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <button id="open-modal">Abrir Modal</button>

    <div id="myModal" class="modal">
        <div class="modal-content">
            <span class="close">&times;</span>
            <h2>Título del Modal</h2>
            <p>Este es el contenido del modal.</p>
        </div>
    </div>

    <script src="scripts.js"></script>
</body>
</html>
```

Figura 1. Componente modal. Código HTML. Fuente: elaboración propia.

```css
body {
    font-family: Arial, sans-serif;
}

.modal {
    display: none;
    position: fixed;
    z-index: 1;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    overflow: auto;
    background-color: rgba(0, 0, 0, 0.4);
    padding-top: 60px;
}

.modal-content {
    background-color: #fefefe;
    margin: 5% auto;
    padding: 20px;
    border: 1px solid #888;
    width: 80%;
    max-width: 500px;
    border-radius: 10px;
    box-shadow: 0 5px 15px rgba(0, 0, 0, 0.3);
}

.close:hover,
.close:focus {
    color: #000;
    text-decoration: none;
}
```

Figura 2. Componente modal. Código CSS. Fuente: elaboración propia.

```javascript
const modal = document.getElementById('myModal');
const btn = document.getElementById('open-modal');
const span = document.getElementsByClassName('close')[0];

btn.onclick = function() {
    modal.style.display = 'block';
}

span.onclick = function() {
    modal.style.display = 'none';
}

window.onclick = function(event) {
    if (event.target == modal) {
        modal.style.display = 'none';
    }
}
```

Figura 3. Componente modal. Código JavaScript. Fuente: elaboración propia.

En este ejemplo se define un botón para abrir el modal y un contenedor modal que contiene el contenido del modal. Este componente modal puede ser reutilizado en cualquier lugar de la aplicación.

El modal tiene un fondo semitransparente que cubre toda la pantalla, y el contenido del modal está centrado con un fondo blanco, bordes redondeados y sombra.

Por último, añadimos «funcionalidad» para abrir y cerrar el modal. Cuando el usuario hace clic fuera del modal o en el botón de cierre, el modal se oculta.

3.3. Integración de menús y cuadros de diálogo

La integración de menús y cuadros de diálogo es un aspecto crucial en el desarrollo de interfaces gráficas de usuario (GUI). Estos elementos no solo enriquecen la experiencia del usuario, sino que también facilitan la interacción con la aplicación, permitiendo acceder a funcionalidades y opciones de manera organizada y controlada.

Menús en aplicaciones GUI

Los menús son una de las formas más comunes de interacción en aplicaciones GUI. En esencia, un **menú** es un **conjunto de opciones** que el usuario puede seleccionar para ejecutar diferentes acciones dentro de la aplicación. Existen, principalmente, **dos tipos de menús:** el menú principal y el menú contextual.

- **Menú principal:** este es el menú que aparece en la **parte superior de la ventana** de la aplicación, generalmente en una barra horizontal. Contiene varias **opciones principales,** como «Archivo», «Editar» y «Ayuda», cada una de las cuales puede desplegar submenús con opciones adicionales. La barra de menú principal siempre está visible y accesible para el usuario, proporcionando un acceso constante a las funcionalidades más importantes de la aplicación.

- **Menú contextual:** a diferencia del menú principal, el menú contextual aparece en **cualquier parte de la ventana,** generalmente cuando el usuario hace clic derecho sobre un elemento específico. Este tipo de menú es útil para ofrecer opciones relacionadas directamente con el elemento seleccionado, como copiar, pegar o eliminar, lo que mejora la eficiencia y la usabilidad al reducir la necesidad de navegar por menús más complejos.

Para crear estos menús en aplicaciones de escritorio utilizando tecnologías como **Swing** en Java, se utilizan clases específicas como `JMenuBar` para la barra de menú, `JMenu` para cada grupo de opciones y `JMenuItem` para las opciones seleccionables dentro de cada menú. También, se pueden usar elementos como `JPopupMenu` para menús contextuales y `JSeparator` para organizar visualmente las opciones dentro de un menú.

Cuadros de diálogo

Los cuadros de diálogo son **ventanas emergentes** que se utilizan en aplicaciones GUI para interactuar con el usuario, ya sea **solicitando información, confirmando** una **acción** o **mostrando mensajes.** Los cuadros de diálogo pueden ser **modales,** lo que significa que bloquean la interacción con la aplicación principal hasta que el usuario haya completado la acción solicitada en el cuadro de diálogo.

- **Cuadros de diálogo predefinidos:** en muchas aplicaciones, se utilizan cuadros de diálogo predefinidos para realizar tareas comunes como mostrar mensajes informativos, solicitar la confirmación del usuario o pedir la entrada de datos. En Java, por ejemplo, la clase `JOptionPane` proporciona métodos como `showMessageDialog` **,** `showConfirmDialog` y `showInputDialog` para crear rápidamente estos tipos de diálogos sin necesidad de diseñarlos desde cero.

- **Cuadros de diálogo personalizados:** sin embargo, en situaciones donde se requiere una funcionalidad específica o un diseño único, los desarrolladores pueden optar por crear cuadros de diálogo personalizados. Esto implica extender la clase `JDialog` y diseñar el cuadro de diálogo de manera similar a como se diseñaría una ventana estándar, incluyendo componentes como botones, campos de texto y etiquetas según las necesidades de la aplicación.

Imaginemos una aplicación de escritorio con una barra de menú que contiene una opción «Archivo» con submenús para «Abrir», «Guardar» y «Salir». Para crear este menú en Java, primero se crea una instancia de `JMenuBar`, luego se añaden las opciones `JMenu` y `JMenuItem`, y, finalmente, se asocian las acciones correspondientes utilizando `ActionListener`.

```
JMenuBar menuBar = new JMenuBar();
JMenu archivoMenu = new JMenu("Archivo");
JMenuItem abrirItem = new JMenuItem("Abrir");
JMenuItem guardarItem = new JMenuItem("Guardar");
JMenuItem salirItem = new JMenuItem("Salir");

archivoMenu.add(abrirItem);
archivoMenu.add(guardarItem);
archivoMenu.addSeparator();
archivoMenu.add(salirItem);
menuBar.add(archivoMenu);

setJMenuBar(menuBar);

salirItem.addActionListener(e -> System.exit(0));
```

Figura 4. Barra de menú. Código Java. Fuente: elaboración propia.

Para crear un cuadro de diálogo simple que solicite confirmación antes de cerrar la aplicación, se puede utilizar `JOptionPane.showConfirmDialog`. Este cuadro de diálogo presenta al usuario opciones como «Sí», «No» y «Cancelar», y devuelve un valor entero que se utiliza para determinar la acción a seguir.

```
int respuesta = JOptionPane.showConfirmDialog(null, "¿Desea salir?", "Confirmación", JOptionPane.YES_NO_OPTION);
if (respuesta == JOptionPane.YES_OPTION) {
    System.exit(0);
}
```

Figura 5. Cuadro de diálogo. Código Java. Fuente: elaboración propia.

A continuación se muestran varios ejemplos de código en Java utilizando la clase JOptionPane para crear diferentes tipos de cuadros de diálogo. Estos ejemplos abarcan los cuadros de diálogo más comunes, incluyendo mensajes de información, confirmación, entrada de datos y opciones personalizadas.

```java
import javax.swing.JOptionPane;

public class InformativeDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Operación completada con éxito.", "Información", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Figura 6. Cuadro de diálogo de mensaje informativo. Código Java. Fuente: elaboración propia.

Este cuadro de diálogo muestra un mensaje simple con un ícono de **información.** El tercer parámetro JOptionPane.INFORMATION_MESSAGE indica el tipo de mensaje a mostrar.

```java
import javax.swing.JOptionPane;

public class WarningDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Está a punto de eliminar un archivo.", "Advertencia", JOptionPane.WARNING_MESSAGE);
    }
}
```

Figura 7. Cuadro de diálogo de advertencia. Código Java. Fuente: elaboración propia.

Este cuadro de diálogo muestra un mensaje con un ícono de **advertencia** para alertar al usuario antes de realizar una acción importante.

```java
import javax.swing.JOptionPane;

public class ErrorDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "Se ha producido un error al guardar el archivo.", "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Figura 8. Cuadro de diálogo de error. Código Java. Fuente: elaboración propia.

Este cuadro de diálogo muestra un mensaje de **error** con un ícono apropiado, notificando al usuario sobre un problema en la aplicación.

```java
import javax.swing.JOptionPane;

public class ConfirmationDialogExample {
    public static void main(String[] args) {
        int respuesta = JOptionPane.showConfirmDialog(null, "¿Está seguro de que desea continuar?", "Confirmación", JOptionPane.YES_NO_OPTION);

        if (respuesta == JOptionPane.YES_OPTION) {
            System.out.println("El usuario eligió 'Sí'.");
        } else {
            System.out.println("El usuario eligió 'No'.");
        }
    }
}
```

Figura 9. Cuadro de diálogo de confirmación. Código Java. Fuente: elaboración propia.

Este tipo de cuadro de diálogo se utiliza para **solicitar** la **confirmación** del usuario antes de proceder con una acción, como cerrar la aplicación o eliminar un archivo.

```java
import javax.swing.JOptionPane;

public class InputDialogExample {
    public static void main(String[] args) {
        String nombre = JOptionPane.showInputDialog(null, "Ingrese su nombre:", "Entrada de Datos", JOptionPane.QUESTION_MESSAGE);

        if (nombre != null) {
            System.out.println("Nombre ingresado: " + nombre);
        } else {
            System.out.println("El usuario canceló la entrada.");
        }
    }
}
```

Figura 10. Cuadro de diálogo de entrada de datos. Código Java. Fuente: elaboración propia.

Un cuadro de diálogo de entrada permite al usuario **introducir datos,** como un nombre o una contraseña.

```java
import javax.swing.JOptionPane;

public class CustomOptionDialogExample {
    public static void main(String[] args) {
        String[] opciones = {"Opción 1", "Opción 2", "Opción 3"};
        int seleccion = JOptionPane.showOptionDialog(null, "Seleccione una opción:", "Opciones Personalizadas",
                JOptionPane.DEFAULT_OPTION, JOptionPane.QUESTION_MESSAGE, null, opciones, opciones[0]);

        System.out.println("Opción seleccionada: " + opciones[seleccion]);
    }
}
```

Figura 11. Cuadro de diálogo con opciones personalizadas. Código Java. Fuente: elaboración propia.

Este cuadro de diálogo muestra una lista de **opciones personalizadas.** El valor devuelto es el índice de la opción seleccionada, que se utiliza para determinar la elección del usuario.

3.4. Propiedades de los componentes más utilizados y su integración

En el desarrollo de interfaces de usuario, los **componentes** son los bloques fundamentales que constituyen una aplicación. Estos componentes tienen **propiedades** que **definen** su **comportamiento, apariencia** y **cómo interactúan** con otros elementos dentro de la aplicación. Comprender las propiedades de los componentes más utilizados y su correcta integración es crucial para el desarrollo eficiente y efectivo de interfaces de usuario.

Propiedades comunes y específicas de los componentes

Los componentes en una interfaz de usuario poseen una serie de **propiedades** que pueden clasificarse en **comunes** y **específicas.** Las **propiedades comunes** son aquellas que la mayoría de los componentes comparten, como la visibilidad, el tamaño, la posición y la capacidad de recibir eventos. Por ejemplo, casi todos los componentes tienen propiedades como `width`, `height`, `visible` y `enabled`, que controlan cómo y cuándo se muestran y utilizan en la interfaz.

Por otro lado, las **propiedades específicas** están relacionadas con la **funcionalidad** particular de un componente. Por ejemplo, un campo de texto (`TextField`) puede tener propiedades como `placeholder`, `maxlength` o `readonly`, que son relevantes únicamente para componentes que permiten la entrada de texto. Otro ejemplo sería un componente `Button`, que podría tener una propiedad `onClick` para definir qué acción realizar cuando el botón es presionado.

Estas propiedades no solo afectan cómo se comporta el componente individualmente, sino que también determinan cómo se **integra** y **comunica** con otros componentes. Por ejemplo, la propiedad `dataSource` en un componente de tabla (`Table`) permite enlazar los datos que se mostrarán en las filas de la tabla, mientras que propiedades como `columnHeaders` controlan la visualización de los encabezados de las columnas.

Integración de componentes en el entorno de desarrollo

La integración de componentes en un entorno de desarrollo no es solo una cuestión de colocarlos en una interfaz. Es fundamental que los desarrolladores comprendan cómo estas propiedades interactúan entre sí y cómo afectan el comportamiento general de la aplicación. Por ejemplo, cuando se trabaja con un entorno de desarrollo integrado (IDE) como Visual Studio o Eclipse, el examinador de propiedades facilita la modificación de estas propiedades de manera visual, permitiendo a los desarrolladores ajustar rápidamente el comportamiento y la apariencia del componente sin necesidad de modificar el código directamente.

Además, los entornos de desarrollo permiten la **categorización** de **propiedades** para facilitar su gestión. Las propiedades pueden ser categorizadas en grupos como «Apariencia», «Comportamiento», «Datos», etc., lo que ayuda a los desarrolladores a localizar y ajustar rápidamente las propiedades relevantes. En muchos casos, las propiedades también vienen con descripciones detalladas que se muestran en el IDE, lo que facilita aún más su configuración correcta.

Persistencia y serialización de componentes

Otro aspecto clave de las propiedades de los componentes es la persistencia y serialización. La **persistencia** se refiere a la capacidad de un **componente** para **mantener** su **estado** a lo largo de diferentes sesiones de la aplicación. Por ejemplo, si un usuario cambia el tamaño de una ventana o el valor de un campo de texto, estos cambios pueden ser persistidos para que la próxima vez que se abra la aplicación, el estado anterior se restaure automáticamente.

L a **serialización,** por otro lado, es el proceso de **convertir** el **estado** de un componente en un **formato** que pueda ser **almacenado** o **transmitido** y luego **reconstruido.** Esto es fundamental en aplicaciones que necesitan guardar el estado del usuario o transmitir datos entre diferentes partes de la aplicación o incluso a través de la red. Existen diferentes métodos de serialización, como la **serialización binaria,** que convierte un objeto en un flujo de *bytes,* o la serialización en XML *(extensible markup language),* que convierte las propiedades públicas de un objeto en un formato legible por humanos que puede ser almacenado o transmitido fácilmente.

Consideremos algunos de los componentes más utilizados y sus propiedades clave:

- **Campos de Texto (** TextField **):**
  - **Propiedades comunes:** text , placeholder , maxlength , enabled .
  - **Propiedades específicas:** password (si el campo debe ocultar el texto), autocomplete (si se permite la autocompletar).

- **Botones (** Button **):**
  - **Propiedades comunes:** text , enabled , visible .
  - **Propiedades específicas:** onClick (acción a ejecutar al hacer clic).
- **Tablas (** Table **):**
  - **Propiedades comunes:** dataSource , columnHeaders , rowHeight .
  - **Propiedades específicas:** sortable (si las columnas se pueden ordenar), filterable (si las filas se pueden filtrar).
- **Listas desplegables (** Select **):**
  - **Propiedades comunes:** options , selectedIndex .
  - **Propiedades específicas:** multiple (si se permite la selección múltiple).

Estos componentes y sus propiedades son fundamentales para construir aplicaciones interactivas y eficientes. Su correcta configuración e integración en el entorno de desarrollo garantiza que la aplicación no solo sea funcional, sino también fácil de usar y mantener.

3.5. Estrategias para la reutilización de código

La reutilización de código es uno de los pilares fundamentales en el desarrollo de *software* eficiente y sostenible. Al adoptar estrategias efectivas para la reutilización de código, los desarrolladores pueden reducir significativamente el tiempo y los costos de desarrollo, mejorar la calidad del *software* y facilitar el mantenimiento a largo plazo. En este contexto, las **estrategias para la reutilización de código** son esenciales para maximizar el valor y la efectividad del trabajo de desarrollo.

Componentización y modularización

Una de las estrategias más importantes para la reutilización de código es la **componentización.** Este enfoque implica dividir una aplicación en **componentes modulares** e **independientes,** cada uno de los cuales encapsula una funcionalidad específica. Los componentes pueden ser **reutilizados** en diferentes partes de la misma aplicación o incluso en proyectos completamente distintos. La **modularización** va de la mano con la componentización, ya que promueve la **organización del código en módulos** que pueden ser combinados de manera flexible para construir aplicaciones completas.

Por ejemplo, en el desarrollo web con *frameworks* como Angular, React o Vue.js, los componentes encapsulan tanto la lógica como la presentación, lo que permite reutilizar botones, formularios y otros elementos UI en múltiples vistas o aplicaciones. Este enfoque no solo facilita la reutilización, sino que también mejora la **mantenibilidad** del código, ya que los desarrolladores pueden actualizar o modificar un componente sin afectar otras partes de la aplicación.

Uso de librerías y frameworks

Otra estrategia clave para la reutilización de código es el uso de librerías y *frameworks.* Las **librerías** son **colecciones** de **funciones, clases** o **componentes** que pueden ser **importadas** y **utilizadas** en diferentes proyectos, mientras que los ***frameworks*** proporcionan un **esqueleto** más estructurado **sobre el cual construir aplicaciones.** Al aprovechar librerías y *frameworks* existentes, los desarrolladores pueden evitar reinventar la rueda, utilizando soluciones probadas y optimizadas para tareas comunes.

Por ejemplo, librerías como lodash en JavaScript ofrecen funciones utilitarias para la manipulación de *arrays,* objetos y *strings,* que pueden ser reutilizadas en diferentes contextos dentro de un proyecto. Los *frameworks,* como Spring en Java o Django en Python, proporcionan una arquitectura robusta que facilita la reutilización de componentes y módulos a través de una estructura bien definida.

Plantillas y generadores de código

Las plantillas y los generadores de código son herramientas poderosas para la reutilización de código, especialmente en proyectos que requieren la creación repetitiva de estructuras similares. Las **plantillas** permiten definir bloques de código que pueden ser reutilizados con diferentes datos o parámetros, mientras que los **generadores de código** automatizan la creación de código repetitivo, reduciendo el esfuerzo manual y minimizando los errores.

Por ejemplo, en el desarrollo de aplicaciones web, los generadores de código como Yeoman pueden crear rápidamente la estructura básica de un proyecto completo, incluyendo la configuración de archivos, componentes iniciales y rutas. Las plantillas de correo electrónico, por otro lado, permiten reutilizar el mismo diseño y estructura para diferentes correos, simplemente reemplazando el contenido específico.

Refactorización y abstracción

Refactorización y abstracción son estrategias que se centran en mejorar la estructura del código existente para facilitar su reutilización. La **refactorización** implica **reescribir** partes del código para hacerlo más modular, legible y reutilizable, sin cambiar su funcionalidad original. Este proceso a menudo incluye la extracción de funciones o métodos comunes en un solo lugar, donde pueden ser reutilizados por diferentes partes del código.

L a **abstracción,** por otro lado, se refiere a la creación de **interfaces** o **clases generales** que encapsulan **comportamientos comunes,** permitiendo que el código específico de la implementación se reutilice en diferentes contextos. Por ejemplo, una clase base `Vehículo` en un sistema de gestión de flotas podría definir propiedades y métodos comunes como arrancar, detener y velocidad, mientras que las clases derivadas como `Auto` y `Camión` podrían reutilizar esta lógica común, añadiendo solo las particularidades específicas de cada tipo de vehículo.

Patrones de diseño

Los patrones de diseño son **soluciones comprobadas** para problemas recurrentes en el desarrollo de *software,* y son una estrategia poderosa para la reutilización de código. Patrones como el Singleton, Fábrica, Decorador y Observador proporcionan una estructura reutilizable para manejar tareas comunes, como la creación de objetos, la modificación de comportamientos o la gestión de eventos.

Por ejemplo, el patrón Fábrica permite la creación de objetos de una manera que abstrae la lógica de instanciación, facilitando la reutilización de código cuando se necesita crear múltiples instancias de una clase con diferentes configuraciones. El patrón Observador es ideal para sistemas donde uno o más objetos necesitan ser notificados de cambios en el estado de otro objeto, lo que es común en aplicaciones con interfaces de usuario interactivas.

Repositorios de código compartido

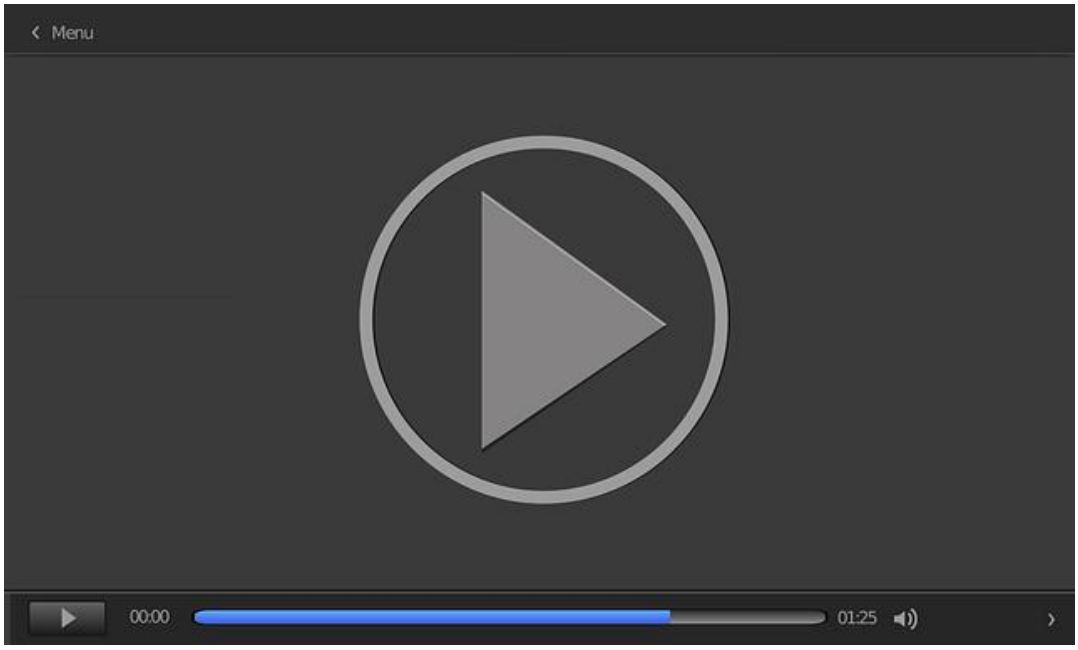
El uso de repositorios de código compartidos es otra estrategia importante para la reutilización de código. Plataformas como GitHub, GitLab o Bitbucket permiten a los equipos de desarrollo **almacenar, compartir** y **colaborar** en código **de manera centralizada.** Estos repositorios actúan como una **base de código común** que puede ser accedida y reutilizada por cualquier miembro del equipo o incluso por desarrolladores de diferentes proyectos.

Los **repositorios de paquetes,** como npm para JavaScript o Maven Central para Java, también son esenciales para la reutilización de código. Estos repositorios permiten a los desarrolladores publicar y distribuir sus librerías y módulos, que luego pueden ser fácilmente integrados en otros proyectos mediante un simple comando de instalación.

Ventanas emergentes en Java (uso de la clase JOptionPane)

Shakmuria. (2017, octubre 28). *Ventanas emergentes en java (Uso de la clase JOptionPane).* [Vídeo]. YouTube. https://www.youtube.com/watch?v=CIKz5eTQkU0

Este vídeo explica la clase JOptionPane y sus métodos como showInputDialog , para mostrar un cuadro de entrada o salida de datos.



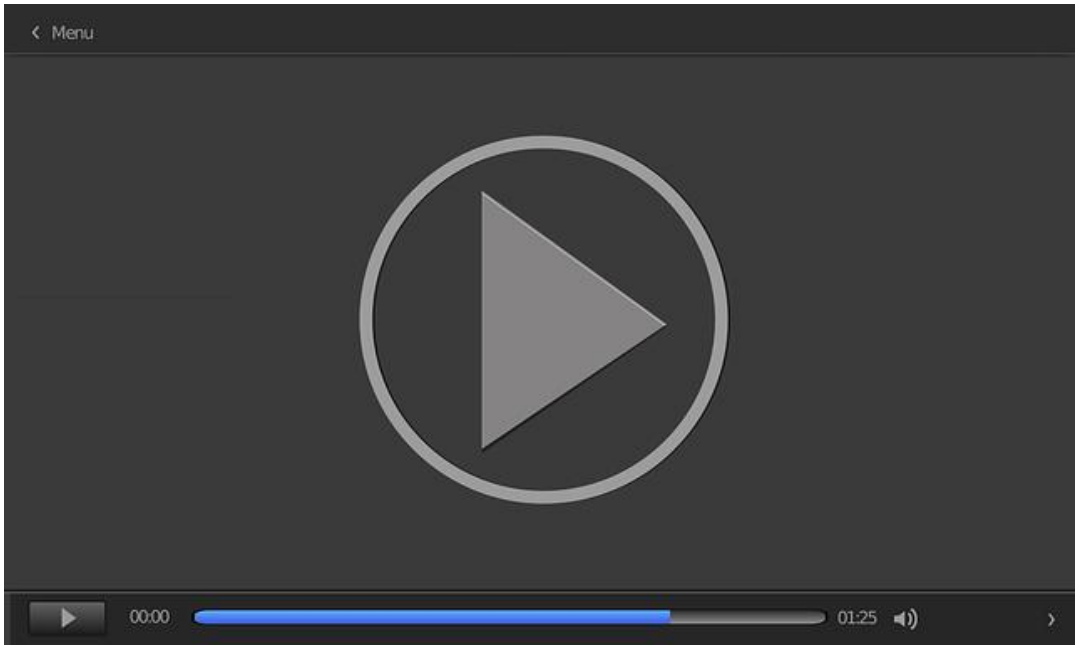
Accede al vídeo:

https://www.youtube.com/embed/CIKz5eTQkU0

Desarrollo de software basado en componentes

Andrés López. (2021, julio 28). *desarrollo de software basado en componentes* [Vídeo]. YouTube. https://www.youtube.com/watch?v=r_C4pKJTU7A

Este vídeo explica el concepto de desarrollo de *software* basado en componentes.



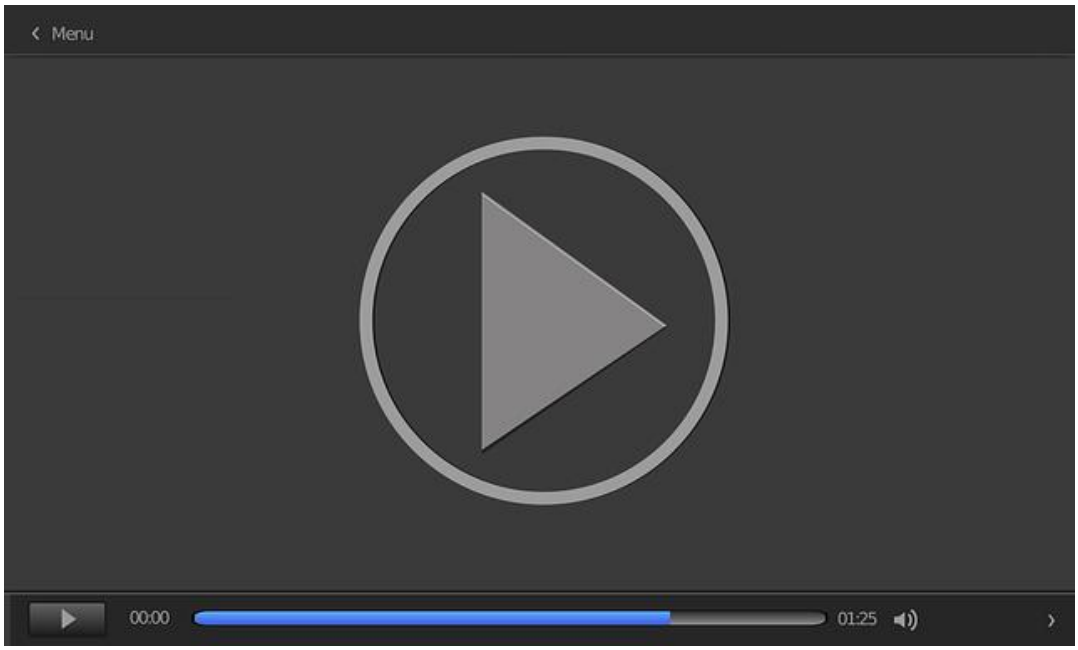
Accede al vídeo:

https://www.youtube.com/embed/r_C4pKJTU7A

Reutilización de software

Rafael Mellado. (2017, mayo 1). Reutilización de *Software* [Vídeo]. YouTube. https://www.youtube.com/watch?v=ToubluB6kis

En este vídeo se explican los principales elementos y definiciones de la reutilización de *software.*



Accede al vídeo:

https://www.youtube.com/embed/ToubluB6kis

Entrenamiento 1

- Planteamiento del ejercicio

Crea un componente de botón reutilizable que pueda ser utilizado en diferentes partes de una aplicación web. El botón debe aceptar propiedades para el texto, el color y la acción que se ejecutará al hacer clic.

- Desarrollo paso a paso

  - Crea un archivo ButtonComponent.html que defina la estructura del botón.
  - Crea un archivo ButtonComponent.css para definir el estilo del botón.
  - Crea un archivo ButtonComponent.js para manejar la funcionalidad del botón.
  - Integra el botón en una página HTML y prueba su funcionamiento con diferentes propiedades.

- Solución

HTML (ButtonComponent.html):

```
<button id="myButton"></button>
```

Figura 12. Solución. Fuente: elaboración propia.

CSS (ButtonComponent.css):

```css
#myButton {
    padding: 10px 20px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    font-size: 16px;
}
```

Figura 13. Solución. Fuente: elaboración propia.

JavaScript (ButtonComponent.js):

```javascript
function createButton(text, color, onClickAction) {
    const button = document.getElementById('myButton');
    button.innerText = text;
    button.style.backgroundColor = color;
    button.onclick = onClickAction;
}
```

Figura 14. Solución. Fuente: elaboración propia.

Integración en una página HTML:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Componente de Botón</title>
    <link rel="stylesheet" href="ButtonComponent.css">
    <script src="ButtonComponent.js" defer></script>
</head>
<body>
    <div>
        <button id="myButton"></button>
    </div>

    <script>
        createButton("Haz clic aquí", "#28a745", function() {
            alert("Botón clicado!");
        });
    </script>
</body>
</html>
```

Figura 15. Solución. Fuente: elaboración propia.

Entrenamiento 2

- Planteamiento del ejercicio

Desarrolla un componente de tarjeta de producto que pueda ser utilizado en una tienda en línea. La tarjeta debe mostrar una imagen, un título, una descripción y un botón de compra.

- Desarrollo paso a paso

  - Crea un archivo ProductCard.html para definir la estructura de la tarjeta.
  - Crea un archivo ProductCard.css para estilizar la tarjeta.
  - Crea un archivo ProductCard.js para configurar las propiedades dinámicas de la tarjeta.
  - Utiliza el componente en una página HTML mostrando varios productos.

- Solución

HTML (ProductCard.html):

```
<div class="product-card">
    <img src="" alt="Imagen del producto" class="product-image">
    <h3 class="product-title"></h3>
    <p class="product-description"></p>
    <button class="buy-button">Comprar</button>
</div>
```

Figura 16. Solución. Fuente: elaboración propia.

CSS (ProductCard.css):

```css
.product-card {
    border: 1px solid #ccc;
    padding: 20px;
    border-radius: 10px;
    text-align: center;
    margin: 10px;
    width: 250px;
}

.product-image {
    width: 100%;
    height: auto;
    border-radius: 5px;
}

.product-title {
    font-size: 18px;
    margin: 10px 0;
}

.product-description {
    font-size: 14px;
    color: #666;
}

.buy-button {
    background-color: #007bff;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
```

Figura 17. Solución. Fuente: elaboración propia.

JavaScript (ProductCard.js):

```javascript
function createProductCard(imageSrc, title, description) {
    const card = document.querySelector('.product-card');
    card.querySelector('.product-image').src = imageSrc;
    card.querySelector('.product-title').innerText = title;
    card.querySelector('.product-description').innerText = description;
}
```

Figura 18. Solución. Fuente: elaboración propia.

Integración en una página HTML:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Tarjeta de Producto</title>
    <link rel="stylesheet" href="ProductCard.css">
    <script src="ProductCard.js" defer></script>
</head>
<body>
    <div class="product-card"></div>

    <script>
        createProductCard("producto1.jpg", "Producto 1", "Descripción breve del producto 1.");
    </script>
</body>
</html>
```

Figura 19. Solución. Fuente: elaboración propia.

Entrenamiento 3

- Planteamiento del ejercicio

Crea un componente modal reutilizable que se pueda utilizar para mostrar información o formularios en diferentes partes de una aplicación.

- Desarrollo paso a paso

  - Crea un archivo ModalComponent.html que contenga la estructura básica del modal.
  - Crea un archivo ModalComponent.css para estilizar el modal.
  - Crea un archivo ModalComponent.js para controlar la apertura y cierre del modal.
  - Integra el modal en una página HTML y prueba su funcionamiento.

- Solución

HTML (ModalComponent.html):

```
<div id="myModal" class="modal">
    <div class="modal-content">
        <span class="close">&times;</span>
        <p>Contenido del modal aquí...</p>
    </div>
</div>
```

Figura 20. Solución. Fuente: elaboración propia.

CSS (ModalComponent.css):

```css
.modal {
    display: none;
    position: fixed;
    z-index: 1;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    overflow: auto;
    background-color: rgba(0,0,0,0.5);
}

.modal-content {
    background-color: white;
    margin: 15% auto;
    padding: 20px;
    border: 1px solid #888;
    width: 80%;
    max-width: 500px;
    border-radius: 10px;
}

.close {
    color: #aaa;
    float: right;
    font-size: 28px;
    font-weight: bold;
    cursor: pointer;
}

.close:hover,
.close:focus {
    color: black;
    text-decoration: none;
    cursor: pointer;
}
```

Figura 21. Solución. Fuente: elaboración propia.

JavaScript (ModalComponent.js):

```
function showModal() {
    const modal = document.getElementById("myModal");
    const span = document.getElementsByClassName("close")[0];

    modal.style.display = "block";

    span.onclick = function() {
        modal.style.display = "none";
    }

    window.onclick = function(event) {
        if (event.target == modal) {
            modal.style.display = "none";
        }
    }
}
```

Figura 22. Solución. Fuente: elaboración propia.

Integración en una página HTML:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Componente Modal</title>
    <link rel="stylesheet" href="ModalComponent.css">
    <script src="ModalComponent.js" defer></script>
</head>
<body>
    <button onclick="showModal()">Abrir Modal</button>
    <div id="myModal" class="modal">
        <div class="modal-content">
            <span class="close">&times;</span>
            <p>Contenido del modal aquí...</p>
        </div>
    </div>
</body>
</html>
```

Figura 23. Solución. Fuente: elaboración propia.

Entrenamiento 4

- Planteamiento del ejercicio

Desarrolla una barra de navegación reutilizable que se pueda utilizar en diferentes páginas de una aplicación web.

- Desarrollo paso a paso

  - Crea un archivo NavbarComponent.html para definir la estructura de la barra de navegación.
  - Crea un archivo NavbarComponent.css para estilizar la barra de navegación.
  - Integra la barra de navegación en una página HTML y ajusta los enlaces.

- Solución

HTML (NavbarComponent.html):

```
<nav class="navbar">
    <ul>
        <li><a href="#home">Inicio</a></li>
        <li><a href="#services">Servicios</a></li>
        <li><a href="#about">Acerca de</a></li>
        <li><a href="#contact">Contacto</a></li>
    </ul>
</nav>
```

Figura 24. Solución. Fuente: elaboración propia.

CSS (NavbarComponent.css):

```css
.navbar {
    background-color: #333;
    overflow: hidden;
}

.navbar ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
    display: flex;
    justify-content: center;
}

.navbar ul li {
    padding: 14px 20px;
}

.navbar ul li a {
    color: white;
    text-decoration: none;
    text-transform: uppercase;
}

.navbar ul li a:hover {
    background-color: #575757;
    border-radius: 4px;
}
```

Figura 25. Solución. Fuente: elaboración propia.

Integración en una página HTML:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Barra de Navegación</title>
    <link rel="stylesheet" href="NavbarComponent.css">
</head>
<body>
    <nav class="navbar">
        <ul>
            <li><a href="#home">Inicio</a></li>
            <li><a href="#services">Servicios</a></li>
            <li><a href="#about">Acerca de</a></li>
            <li><a href="#contact">Contacto</a></li>
        </ul>
    </nav>
</body>
</html>
```

Figura 26. Solución. Fuente: elaboración propia.

Entrenamiento 5

- Planteamiento del ejercicio

Crea un componente de formulario reutilizable que pueda ser utilizado para capturar datos del usuario. El formulario debe incluir campos de entrada de texto, un área de texto y un botón de envío.

- Desarrollo paso a paso

  - Crea un archivo FormComponent.html para definir la estructura del formulario.
  - Crea un archivo FormComponent.css para estilizar el formulario.
  - Crea un archivo FormComponent.js para manejar la validación básica del formulario.
  - Integra el formulario en una página HTML y prueba su funcionalidad.

- Solución

HTML (FormComponent.html):

```
<form id="myForm">
    <label for="name">Nombre:</label>
    <input type="text" id="name" name="name"><br><br>
    <label for="email">Correo Electrónico:</label>
    <input type="email" id="email" name="email"><br><br>
    <label for="message">Mensaje:</label><br>
    <textarea id="message" name="message"></textarea><br><br>
    <button type="submit">Enviar</button>
</form>
```

Figura 27. Solución. Fuente: elaboración propia.

CSS (FormComponent.css):

```css
form {
    max-width: 400px;
    margin: 0 auto;
    padding: 20px;
    border: 1px solid #ccc;
    border-radius: 10px;
    background-color: #f9f9f9;
}

label {
    font-weight: bold;
}

input, textarea {
    width: 100%;
    padding: 10px;
    margin: 5px 0;
    border: 1px solid #ccc;
    border-radius: 5px;
}

button {
    background-color: #28a745;
    color: white;
    padding: 10px 15px;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}
```

Figura 28. Solución. Fuente: elaboración propia.

JavaScript (FormComponent.js):

```javascript
document.getElementById('myForm').addEventListener('submit', function(event) {
    event.preventDefault();
    const name = document.getElementById('name').value;
    const email = document.getElementById('email').value;
    const message = document.getElementById('message').value;

    if (name === "" || email === "" || message === "") {
        alert("Todos los campos son obligatorios.");
    } else {
        alert("Formulario enviado con éxito!");
    }
});
```

Figura 29. Solución. Fuente: elaboración propia.

Integración en una página HTML:

```html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Formulario Reutilizable</title>
    <link rel="stylesheet" href="FormComponent.css">
    <script src="FormComponent.js" defer></script>
</head>
<body>
    <form id="myForm">
        <label for="name">Nombre:</label>
        <input type="text" id="name" name="name"><br><br>
        <label for="email">Correo Electrónico:</label>
        <input type="email" id="email" name="email"><br><br>
        <label for="message">Mensaje:</label><br>
        <textarea id="message" name="message"></textarea><br><br>
        <button type="submit">Enviar</button>
    </form>
</body>
</html>
```

Figura 30. Solución. Fuente: elaboración propia.