

TEMA 3 - COMPONENTES DE USUARIO

Índice de contenidos

1. Personalizar la apariencia de los componentes.
2. Derivar de un componente existente.
3. Crear un componente completamente nuevo.
4. Utilización de nuevos componentes en aplicaciones.

Introducción

En cualquier tecnología para crear interfaces (Qt, JavaFX, Android, .NET...) solemos disponer de un catálogo amplio de controles: botones, cajas de texto, listas, cuadros combinados...

Aun así, hay proyectos donde necesitamos un componente que **no viene incluido de forma nativa**.

Cuando eso ocurre, normalmente tenemos **cuatro caminos posibles**:

1. Buscar un componente creado por terceros.
2. Personalizar la apariencia de uno existente.
3. Derivar de un componente ya disponible.
4. Crear uno completamente nuevo.

Cada opción tiene ventajas y límites, así que conviene elegir según lo que necesitemos.

¿Sabías que...?

En Qt el catálogo de componentes es muy grande. Por eso no hay tanta oferta de controles de terceros comparado con otras tecnologías. Aun así, encontrarás colecciones en GitHub, foros y páginas especializadas que pueden darte ideas o servir como base.

Personalizar la apariencia de un componente

Muchísimas veces no necesitamos un componente nuevo.

La funcionalidad ya existe, solo queremos que se vea diferente.

Un ejemplo muy típico es el **interruptor de activación (switch)** que vemos en móviles:

- **Funcionalidad:** actúa como un checkbox (activado / desactivado).
- **Apariencia:** un pequeño interruptor que se desliza.



Esto se consigue aplicando **estilos, temas** o dibujando nosotros mismos la apariencia del componente.

Ejemplo

Un switch y un checkbox hacen lo mismo, pero el aspecto cambia totalmente gracias a la personalización del estilo.

Derivar de un componente existente

Cuando necesitamos un comportamiento parecido al de un componente que ya existe, una opción muy habitual es **heredar** del control original y añadir lo que falte:

- Nuevas propiedades.
- Nuevas señales y ranuras.
- Comportamientos personalizados.

En ocasiones conviene derivar de una clase más general, no del control final.

Ejemplo en Qt

La clase **QAbstractButton** es la base de todos los botones:

QPushButton, QCheckBox, QRadioButton, QToolButton...

Si quieres crear un botón personalizado, puedes partir de **QAbstractButton** y así controlar mejor su comportamiento.

Crear un componente completamente nuevo

A veces, lo que necesitamos **no se parece en nada** a ningún componente disponible.

En ese caso no queda más remedio que crearlo desde cero.

En Qt esto implica:

- Heredar de **QWidget**, punto de partida de casi todos los elementos visuales.
- Dibujar el componente con **QPainter**, que permite pintar formas, textos, bordes y cualquier representación visual.

¿Quieres saber más?

QPainter funciona como un "lienzo digital" donde tú decides qué dibujar en cada repaintado.

Se utiliza mucho en:

- componentes de gráficos personalizados,
- indicadores visuales,
- diagramas e infografías,
- controles totalmente propios.

1. Personalizar la apariencia del componente

En muchas ocasiones, el componente que necesitamos **ya existe** en la biblioteca de Qt: botones, casillas de verificación, cuadros de texto, listas...

Funciona perfectamente, pero **no tiene el aspecto visual que queremos** o que exige el diseño del proyecto. Antes de crear un componente nuevo, es conveniente preguntarse:

"¿La funcionalidad me sirve tal cual?"

¿Solo necesito que se vea diferente?"

Si la respuesta es sí, entonces lo más adecuado es **personalizar la apariencia** mediante **estilos QSS**, el sistema de estilos de Qt que funciona de forma muy parecida al CSS de las páginas web.

1.1. ¿Qué significa personalizar un componente con estilos?

Significa modificar únicamente **su aspecto visual**, sin cambiar su comportamiento.

El componente seguirá funcionando igual que siempre, pero se verá distinto.

Es útil para cambiar:

- colores
- bordes
- esquinas redondeadas
- colores al pasar el ratón
- iconos
- tamaños mínimos
- efectos visuales simples

Todo esto se hace **sin tocar su lógica interna**.

Ejemplo de idea general:

- Un **QCheckBox** puede seguir funcionando como casilla de verificación,
- pero puede parecer un interruptor deslizante gracias a un estilo que modifique su apariencia.

Ejemplo: interruptor (switch) vs. checkbox

En dispositivos móviles vemos interruptores tipo "switch": un pequeño control deslizante que cambia de lado cuando lo activamos.



Funcionalmente:

- tiene dos estados (activado / desactivado),
- emite una señal cuando cambia,
- permite consultar su valor.

Esto es **exactamente lo que hace un QCheckBox**.

La diferencia está **solo en la apariencia**.

Por eso, para crear un interruptor estilo móvil, no hace falta programar nada nuevo: basta con aplicar una hoja de estilo al **QCheckBox**.

1.2. Qué es QSS (Qt Style Sheets)

Qt permite aplicar estilos a los widgets mediante QSS, un lenguaje muy inspirado en CSS.

Podemos definir estilos:

- para un tipo de componente (por ejemplo, todos los **QPushButton**)
- para un componente específico
- para estados concretos (hover, checked, deshabilitado...)
- para partes internas del widget (como el indicador de un checkbox)

Ejemplo:

```
1 QPushButton {  
2     background-color: #4CAF50;  
3     color: white;  
4     border-radius: 8px;  
5     padding: 6px 12px;  
6 }
```

Este estilo convierte todos los botones en botones “planos” con esquinas redondeadas.

En qué se parece a CSS

Se parece a CSS en:

- **Sintaxis:**

```

1 QPushButton {
2     background-color: #3498db;
3     color: white;
4 }
```

- **Uso de selectores:**

- Por tipo: `QPushButton`, `QLineEdit`...
- Por id: `#miBoton`
- Por estados: `QPushButton:hover`

- **Uso de propiedades conocidas:**

`color`, `background-color`, `border`, `font-size`, etc.

En qué NO funciona igual que CSS

1. **La herencia no es como en CSS**

En CSS, si pones `color: red` en el `body`, normalmente todo el texto hereda el color.

En QSS, si pones:

```

1 QMainWindow {
2     background-color: red;
3 }
```

solo el `QMainWindow` cambia.

Los widgets dentro siguen con su estilo por defecto, salvo que también los incluyas en la hoja de estilo.

2. **No todos los widgets aceptan todas las propiedades**

Algunos estilos simplemente no tienen efecto según el widget o el estilo de sistema (Windows, macOS...).

3. **Depende del estilo de Qt**

Con el estilo `"Fusion"` QSS funciona mejor.

```

1 app.setStyle("Fusion")
```

► **¿Qué es un estilo en Qt?**

Dónde se puede aplicar QSS

En PySide6 puedes aplicar estilos en tres niveles:

1. A un widget concreto

```
1  boton.setStyleSheet("background-color: red;")
```

2. A una ventana (y sus hijos)

Los estilos se tienen en cuenta para los widgets que están dentro, si los selectores coinciden.

```
1  self.setStyleSheet("""  
2      QPushButton { background-color: red; }  
3  """")
```

3. A toda la aplicación

```
1  app.setStyleSheet("""  
2      QPushButton { background-color: red; }  
3  """")
```

En este caso, todos los `QPushButton` de la app usarán ese estilo, salvo que se sobrescriba en una ventana o en un widget concreto.

Cómo se integra CSS en PySide6

Qt permite personalizar la apariencia de los widgets mediante **QSS (Qt Style Sheets)**, un sistema muy similar al CSS de las páginas web.

Estos estilos se pueden aplicar de varias formas:

- directamente sobre un widget usando `setStyleSheet`,
- aplicándolos a toda la aplicación mediante `app.setStyleSheet`,
- definiendo estilos mediante nombres (`objectName`),
- o cargándolos desde un fichero externo `.qss`.

Todas estas opciones son válidas, pero para mantener el código limpio y facilitar el trabajo en clase, **nos centraremos únicamente en la forma más organizada y fácil de mantener**:

Cargar un archivo `.qss` desde Python y aplicarlo a toda la aplicación.

Este método permite separar claramente:

- el **código Python** → lo que hace la aplicación,
- el **archivo QSS** → cómo se ve la aplicación.

Así conseguimos que el alumnado se centre primero en la lógica de interfaz y pueda modificar estilos sin riesgo de romper el programa.

Cargar QSS desde un fichero `.qss`

Esta es la forma más ordenada para proyectos medianos o grandes.

En lugar de escribir el QSS dentro del código, lo guardamos en un fichero separado, por ejemplo `estilos.qss`.

¿Qué es un fichero `.qss`?

Un fichero `.qss` es un archivo de texto que contiene reglas de estilo aplicables a los widgets.

Dentro de él podemos definir:

- colores de fondo,
- colores de texto,
- bordes,
- esquinas redondeadas,
- tamaños de letra,
- separaciones,
- apariencia de botones, etiquetas, checkbox, etc.

Ejemplo de contenido de un fichero `estilos.qss`:

```
1 QPushButton {  
2     background-color: #4CAF50;  
3     color: white;  
4     border-radius: 8px;  
5 }  
6  
7 QLabel {  
8     font-size: 14px;  
9 }
```

Pasos para usar un fichero `.qss` en PySide6

Siempre seguiremos estos pasos:

1. Crear el fichero `estilos.qss` en la misma carpeta que el programa.
2. Escribir las reglas de estilo dentro del archivo.
3. Abrir el fichero desde Python con `open`.
4. Leer su contenido con `read()`.
5. Aplicar los estilos con `app.setStyleSheet`.
6. Crear la ventana y los widgets normalmente.

Ejemplo

Fichero `estilos.qss`

```
1 QPushButton {  
2     background-color: #4CAF50;  
3     color: white;  
4     border-radius: 8px;  
5     padding: 6px 12px;  
6 }  
7  
8 QLabel {  
9     font-size: 14px;  
10 }
```

Código Python

```
1  from PySide6.QtWidgets import QApplication, QWidget, QPushButton
2  import sys
3
4  app = QApplication(sys.argv)
5
6  # Cargamos y aplicamos el fichero de estilos.
7  with open("estilos.qss", "r") as f:
8      app.setStyleSheet(f.read())
9
10 ventana = QWidget()
11 ventana.setWindowTitle("Ejemplo con estilos QSS")
12
13 layout = QVBoxLayout()
14
15 boton = QPushButton("Aceptar")
16 label = QLabel("Texto con estilo")
17
18 layout.addWidget(label)
19 layout.addWidget(boton)
20 ventana.setLayout(layout)
21
22 ventana.show()
23 app.exec()
```

- `app.setStyleSheet(...)` aplica el estilo **a toda la aplicación**.
- Cualquier widget que coincida con las reglas del `.qss` se verá afectado.
- Para cambiar el diseño solo hay que modificar el archivo `estilos.qss`.

Aplicar estilos básicos

Los primeros estilos que trabajaremos son:

- `color` → color del texto
- `background-color` → fondo del widget
- `font-size` y `font-weight` → tamaño y grosor de letra
- `border` → borde completo
- `border-radius` → esquinas redondeadas
- `padding` → espacio interno entre el contenido y el borde del widget

Ejemplo

```
1 QLineEditText {
2     background-color: rgb(255, 200, 200);
3     border: 2px solid red;
4     border-radius: 8px;
5     font-size: 14px;
6     padding: 6px;
7 }
```

Ejemplo completo, básico y sencillo con QSS

Vamos a hacer una app pequeña con:

- un `QLabel`
- un `QLineEdit`
- dos `QPushButton`

Aplicaremos QSS a nivel de ventana (`self.setStyleSheet(...)`).

Código completo

```
1  from PySide6.QtWidgets import (
2      QApplication,
3      QMainWindow,
4      QWidget,
5      QVBoxLayout,
6      QLabel,
7      QLineEdit,
8      QPushButton
9  )
10
11
12 class MainWindow(QMainWindow):
13     def __init__(self):
14         super().__init__()
15
16         self.setWindowTitle("Ejemplo de QSS")
17
18         # --- Widgets ---
19         self.label = QLabel("Introduce tu nombre:")
20         self.input_nombre = QLineEdit()
21         self.boton_aceptar = QPushButton("Aceptar")
22         self.boton_cancelar = QPushButton("Cancelar")
23
24         # --- Layout ---
25         layout = QVBoxLayout()
26         layout.addWidget(self.label)
27         layout.addWidget(self.input_nombre)
28         layout.addWidget(self.boton_aceptar)
29         layout.addWidget(self.boton_cancelar)
30
31         contenedor = QWidget()
32         contenedor.setLayout(layout)
33         self.setCentralWidget(contenedor)
34
35         # --- Hoja de estilos QSS aplicada a la ventana ---
36         estilos = """
```

Qué se ve en la interfaz

- Una ventana con fondo gris claro.
- Una etiqueta con texto en color oscuro y tamaño algo grande.
- Una caja de texto con borde azul y esquinas redondeadas.
- Dos botones azules con texto blanco.
- Efecto visual al pasar el ratón por encima y al pulsar los botones.

1.3. Propiedades básicas de estilo en QSS

QSS proporciona un conjunto amplio de propiedades para controlar la apariencia de los widgets de Qt/PySide6.

Estas propiedades permiten modificar colores, fondos, bordes, tipografía, espacios internos y externos, iconos y otros elementos visuales.

Constituyen la base del diseño visual antes de trabajar con el *box model*, los selectores o los pseudo-estados.

Color

La propiedad `color` establece el color del texto de un widget.

```
1 QLabel {  
2     color: #2c3e50;  
3 }
```

Formas habituales de definir colores:

- nombres estándar: `red`, `white`, `black`...
- formato RGB: `rgb(255, 180, 120)`
- formato RGBA: `rgba(255, 180, 120, 180)`
- formato hexadecimal: `#3498db`, `#e74c3c`

Fondo (*background*)

El fondo se controla con diversas propiedades.

Color de fondo:

```
1 QWidget {  
2     background-color: #f0f0f0;  
3 }
```

Imagen como fondo:

```
1 QLabel {  
2     background-image: url("fondo.png");  
3 }
```

Repetición de imagen:

```
1 background-repeat: repeat; /* repeat-x | repeat-y | no-repeat | space
```

Posición de la imagen:

1

```
background-position: center; /* top, bottom, left, right, top
```

Ajustes más técnicos:

1

```
background-origin: padding; /* margin | border | padding | content */  
background-clip: border;
```

Bordes

La familia `border` controla la línea que rodea al widget.

Borde básico:

1

```
QPushButton {  
    border: 2px solid #2980b9;  
}
```

Incluye: grosor, estilo y color.

Estilos posibles: `solid`, `dashed`, `dotted`, `double`, `inset`, `outset`, `groove`, `ridge`, `none`.

Bordes por lado:

1

```
border-top: 2px solid red;
```

Esquinas redondeadas:

1

```
border-radius: 10px;
```

Imagen como borde:

1

```
border-image: url("marco.png") 5 5 5 5 stretch;
```

Tipografía (*font*)

Propiedades que modifican el texto del widget.

```
1 QLabel {  
2     font-size: 16px;  
3     font-weight: bold;  
4     font-style: italic;  
5     font-family: "Arial";  
6 }
```

También existe la forma abreviada:

```
1 font: italic bold 16px "Segoe UI";
```

Padding (espacio interno)

Crea espacio entre el contenido y el borde.

```
1 QLineEdit {  
2     padding: 6px;  
3 }
```

O en formato individual por lados:

```
1 padding: 4px 8px 4px 8px; /* arriba, derecha, abajo, izquierda
```

Margin (espacio externo)

Define la separación entre un widget y los elementos que lo rodean.

```
1 margin: 10px;
```

Formato por lados:

```
1 margin: 5px 10px 5px 10px;
```

Iconos

Algunos widgets aceptan iconos mediante propiedades Qt expuestas a QSS.

```

1 QPushButton {
2     qproperty-icon: url("guardar.png");
3     qproperty-iconSize: 24px 24px;
4 }
```

Es posible definir iconos distintos según el estado:

```

1 QPushButton:pressed {
2     qproperty-icon: url("icono_pulsado.png");
3 }
```

Alineación interna

Algunos widgets permiten establecer la alineación del contenido:

```
1 qproperty-alignment: AlignCenter;
```

Otros valores: AlignLeft, AlignRight, AlignTop, AlignBottom, AlignHCenter, AlignVCenter.

Otras propiedades útiles

Tamaño mínimo y máximo:

```

1 min-width: 100px;
2 max-height: 150px;
```

Colores de selección de texto:

```

1 selection-background-color: yellow;
2 selection-color: black;
```

Unidades habituales:

px (píxeles), pt (puntos).

1.4. Selectores QSS (cómo apuntar a un widget)

Una hoja de estilo QSS se aplica a los widgets que **coinciden con un selector**, es decir, la parte que va antes de las llaves:

```
1 QPushButton {  
2     background-color: orange;  
3 }
```

Qt busca todos los widgets `QPushButton` y aplica el estilo.

Los selectores permiten:

- aplicar estilos a todos los widgets de un tipo,
- limitar estilos a ciertas zonas,
- evitar repetir reglas
- controlar mejor la apariencia de la interfaz.

Tipos principales de selectores:

- por tipo
- universal
- por descendencia
- listas de selectores
- subcontroles (`::indicator`, `::handle`, etc.)

Selector por tipo

Es el más usado:

```
1 QPushButton {  
2     background-color: orange;  
3 }
```

Afecta a todos los widgets de ese tipo (`QPushButton`, `QLabel`, `QLineEdit`, etc.) y también a clases hijas.

Sirve para dar una apariencia coherente a toda la aplicación.

Selector universal (*)

```
1 * {
2     font-family: "Segoe UI";
3     font-size: 14px;
4 }
```

Afecta a todos los widgets que acepten esas propiedades.

Es útil para definir una fuente o tamaño de letra base, pero conviene usarlo con moderación.

Selectores por descendencia

Aplican estilos solo a widgets que están **dentro de otros**:

```
1 QMainWindow QPushButton {
2     background-color: #3498db;
3 }
4
5 QGroupBox QLabel {
6     font-weight: bold;
7 }
```

Permiten cambiar la apariencia según el contenedor (por ejemplo, solo los botones dentro de una ventana o las etiquetas dentro de un **QGroupBox**).

Listas de selectores

Permiten aplicar el mismo estilo a varios tipos:

```
1 QPushButton, QToolButton {
2     background-color: #3498db;
3     color: white;
4 }
```

```
1 QLineEdit, QTextEdit, QPlainTextEdit {  
2     background-color: #fdfdfd;  
3     border: 1px solid #bdc3c7;  
4 }
```

Evitan duplicar reglas y ayudan a mantener consistencia entre widgets similares.

Subcontroles

Algunos widgets tienen partes internas que se estilizan con `:::`:

- `QCheckBox::indicator` → casilla
- `QRadioButton::indicator` → círculo
- `QScrollBar::handle` → tirador
- `QSlider::handle` → deslizador

Ejemplo:

```
1 QCheckBox::indicator {  
2     width: 20px;  
3     height: 20px;  
4     border: 1px solid #7f8c8d;  
5     background-color: white;  
6     border-radius: 3px;  
7 }
```

Especificidad y prioridad

Cuando varias reglas afectan al mismo widget:

- gana el **selector más específico**:

```

1 QPushButton {
2     background-color: #3498db;
3 }
4
5 QMainWindow QPushButton {
6     background-color: #e74c3c;
7 }

```

Aquí, los botones dentro de `QMainWindow` quedan en rojo.

Además, las reglas que aparecen **más abajo** en la hoja pueden sobrescribir a otras con la misma especificidad.

Buenas prácticas

- Empezar usando sobre todo **selectores por tipo**.
- Usar `*` solo para estilos muy generales.
- Usar subcontroles para personalizar partes concretas (`::indicator`, `::handle`...).
- Evitar jerarquías muy profundas y difíciles de leer.

1.5. Pseudo-selectores (estados de los widgets)

Los pseudo-selectores permiten cambiar el estilo según el **estado** del widget:

- ratón encima (`::hover`)
- pulsado (`::pressed`)
- marcado (`::checked`)
- deshabilitado (`::disabled`)
- con foco de teclado (`::focus`)

Se añaden con `:` al selector base:

```

1 QPushButton:hover { ... }
2 QCheckBox:checked { ... }
3 QLineEdit:disabled { ... }

```

:hover

Ratón encima del widget:

```
1 QPushButton {  
2     background-color: #3498db;  
3     color: white;  
4 }  
5  
6 QPushButton:hover {  
7     background-color: #1abc9c;  
8 }
```

:pressed

Botón pulsado:

```
1 QPushButton:pressed {  
2     background-color: #2980b9;  
3     padding-top: 2px;  
4     padding-left: 2px;  
5 }
```

Simula que el botón se hunde ligeramente.

:checked

Marcado en **QCheckBox**, **QRadioButton** o botones con
`setCheckable(True)`:

```
1 QCheckBox::indicator:checked {  
2     background-color: #2ecc71;  
3 }
```

:disabled

Widget deshabilitado:

```

1 QPushButton::disabled {
2     background-color: #bdc3c7;
3     color: #7f8c8d;
4 }
```

Apariencia "apagada".

:focus

Widget con foco de teclado:

```

1 QLineEdit:focus {
2     border: 2px solid #2980b9;
3     background-color: #ecf6fb;
4 }
```

Ayuda a localizar el campo activo.

Combinaciones, negación y otros estados

Es posible combinar estados:

```

1 QCheckBox::indicator:checked:hover {
2     background-color: #2ecc71;
3 }
```

Y negarlos con **:!estado**:

```

1 QPushButton:!hover {
2     background-color: #3498db;
3 }
```

Otros pseudo-selectores útiles: **:read-only**, **:on**, **:off**, **:open**,
:selected, **:horizontal**, **:vertical**.

En conjunto, los pseudo-selectores permiten que la interfaz responda visualmente a las acciones del usuario (pasar el ratón, pulsar, activar, desactivar, etc.).

2. Derivar de un componente existente

En el desarrollo de interfaces con Qt es muy frecuente que necesitemos un componente que **no existe exactamente** en la biblioteca estándar, pero que sí es **muy similar** a uno de los widgets ya incluidos.

En esos casos, crear un componente desde cero no suele ser la mejor opción. Es mucho más práctico **derivar** (heredar) de un componente ya existente y extenderlo con lo que necesitamos. Esta estrategia aparece en muchos contextos de desarrollo, y en Qt resulta especialmente útil porque la mayoría de widgets ya vienen equipados con un sistema muy completo de señales, eventos, propiedades y comportamiento visual.

2.1. ¿Qué significa realmente "derivar" de un componente?

Cuando derivamos de un widget de Qt, estamos creando una **nueva clase** que:

1. **Parte** del comportamiento ya implementado en la clase base.
2. **Añade o modifica** aquello que necesitamos.
3. Mantiene todas las capacidades del widget original:
 - integración en layouts,
 - gestión de foco,
 - eventos,
 - señales estándar,
 - propiedades accesibles desde Qt Designer.

De esta forma, el nuevo componente se puede utilizar en cualquier formulario igual que si fuese un widget de Qt, pero con funcionalidades propias.

Comparación intuitiva

Imagina que Qt fuese un "supermercado de widgets":

- Hay widgets que te encajan perfectamente.
- Otros casi encajan, pero les falta un detalle.
- Derivar sería como tomar un producto y personalizarlo sin tener que fabricarlo desde cero.

¿Por qué derivar?

Qt está diseñado para que derivar no sea solo conveniente, sino también natural.

Reducción de complejidad

Los widgets de Qt ya incluyen:

- Control del foco en teclado.
- Gestión del repaintado.
- Integración con el sistema operativo.
- Señales muy utilizadas (clicked, textChanged...).
- Atajos de teclado.
- Cambios de estado visual.

Reescribir todo esto sería tedioso y propenso a errores.

Reutilización de comportamiento probado

Los widgets de Qt pasan por procesos exhaustivos de prueba.

Cuando heredas de ellos:

- Sabes que responden bien a distintos sistemas operativos.
- Sabes que funcionan con distintos temas y estilos.
- Sabes que respetan la accesibilidad.

Uniformidad en la interfaz

Si todos los widgets provienen de la misma arquitectura:

- La aplicación es más predecible.
- Las interacciones son coherentes.
- El código es más fácil de mantener.

2.2. Elegir correctamente la clase base

Una de las claves del éxito al derivar es escoger bien **de qué clase heredar**.

A veces se elige demasiado arriba en la jerarquía (como `QWidget`), lo que implica tener que implementar más cosas de las necesarias.

Otras veces se elige demasiado abajo y el widget base aporta un comportamiento que estorba.

Vamos a profundizar en las opciones.

1. Heredar de `QWidget`

`QWidget` es la base de casi todo en Qt, pero:

- No tiene comportamiento especializado.
- No tiene señales propias para texto, clics o cambios de estado.
- Es un lienzo en blanco.

Usarlo tiene sentido cuando el componente que quieras crear es **completamente nuevo**, como:

- un control gráfico de tipo dial personalizado,
- una barra de progreso decorativa,
- un widget que representa datos de una gráfica.

Ventaja: máximo control.

Inconveniente: exige implementar más lógica.

2. Heredar de un widget concreto

Por ejemplo:

Heredar de `QPushButton`

Conviene cuando el componente:

- se pulsa,
- emite clics,

- muestra texto o ícono,
- responde a la interacción como un botón normal.

Cambiar comportamiento es sencillo:

- modificar qué ocurre al pulsar,
- alterar su aspecto,
- añadir estados adicionales,
- conectar nuevas señales.

Heredar de `QLabel`

Ideal para:

- etiquetas dinámicas,
- indicadores visuales,
- etiquetas que cambian al pasar el ratón,
- etiquetas que actúan como enlaces.

`QLabel` ya se pinta sola y gestiona imágenes, texto y estilos.

Heredar de `QLineEdit`

Perfecto para:

- validaciones personalizadas,
- avisos visuales,
- filtros de entrada,
- autocompletado especializado.

3. Heredar de clases más abstractas

`QAbstractButton`

Se usa cuando queremos:

- un botón con comportamiento propio,
- un aspecto completamente distinto,
- un widget que actúe como botón pero no se parezca a uno.

Proporciona:

- estados pulsado/marcado,
- señales `clicked`, `pressed`, `released`,

- propiedades básicas.

Aquí, el programador suele sobreescibir **paintEvent** y dibuja el botón a mano.

¿Cuándo elegir una clase abstracta?

Cuando el widget concreto aporta un comportamiento que:

- te estorbará,
- cambiarás completamente,
- te obliga a "luchar contra él".

Comprendiendo la jerarquía: caso de **QAbstractButton**

La jerarquía típica es:

```
1 QWidget
2   └── QAbstractButton
3     ├── QPushButton
4     ├── QCheckBox
5     ├── QRadioButton
6     └── QToolButton
```

Elegir el punto adecuado de esta jerarquía determina:

- cuánto código reutilizas,
- cuánto control tienes,
- qué señales obtienes "gratis".

Por ejemplo, si necesitas:

- un botón circular,
- o un botón con animaciones,
- o un botón que cambia dinámicamente de forma,

heredar de **QPushButton** implica tener que desactivar estilos y comportamientos; mientras que con **QAbstractButton** tienes libertad total.

Ejemplos

Botón contador

```
1  from PySide6.QtWidgets import QPushButton
2
3  class BotonContador(QPushButton):
4      def __init__(self, parent=None):
5          texto_inicial = "Pulsado 0 veces"
6          super().__init__(texto_inicial, parent)
7          self.__contador = 0
8          self.clicked.connect(self._incrementar)
9
10     def _incrementar(self):
11         self.__contador = self.__contador + 1
12         nuevo_texto = "Pulsado " + str(self.__contador) + " veces"
13         self.setText(nuevo_texto)
14
15     class VentanaPrincipal(QMainWindow):
16         def __init__(self):
17             super().__init__()
18             self.setWindowTitle("Prueba de BotonContador")
19
20         # Creamos el botón personalizado
21         boton = BotonContador(self)
22
23         # Lo establecemos como widget central de la ventana
24         self.setCentralWidget(boton)
25
26
27     app = QApplication([])
28     ventana = VentanaPrincipal()
29     ventana.show()
30     app.exec()
```

- `__contador` mantiene el estado interno del botón.
- `clicked` es una señal ya implementada por Qt.
- La interfaz no cambia: desde fuera sigue siendo un botón estándar.



Con uso de `palette()`

1. ¿Qué es `palette()`? · En PySide6, cada widget (botones, cajas de texto, ventanas...

2.3. Señales personalizadas

¿Qué es una señal en Qt?

En Qt, una **señal** es una forma de avisar de que *algo ha ocurrido* en un widget o componente.

Ejemplos que ya conocéis:

- `clicked` en un botón
- `textChanged` en un `QLineEdit`
- `valueChanged` en un `QSlider`

Cuando una señal se emite, Qt puede activar uno o varios **slots** (funciones conectadas a esa señal).

Puedes imaginarlo así:

Una señal es como decir "Ey, acabo de cambiar. Si alguien está escuchando, que reaccione".

¿Por qué crear señales personalizadas?

Cuando derivamos un widget, a veces queremos que **nuestro componente avise** de un cambio interno importante.

Por ejemplo:

- Un semáforo que cambia de color.
- Un widget que valida texto.
- Un contador que alcanza cierto número.
- Un control de progreso "especial" que entra en estado crítico.

Las señales propias permiten que otros widgets **reaccionen** sin necesidad de que conozcan su implementación interna.

Cómo se declara una señal propia

Sintaxis básica

```
1 from PySide6.QtCore import Signal
```

Y dentro de la clase:

```
1 class MiWidget(QWidget):  
2     valor_actualizado = Signal(int) # La señal enviará un ente
```

- Las señales **siempre** se declaran como **atributos de clase**, nunca en `__init__`.
- Pueden enviar parámetros (enteros, cadenas, objetos...).
- Sus nombres se escriben en minúscula y estilo “snake_case” por convención, pero Qt no obliga.

Cómo emitir una señal

Para emitir una señal se usa el método `.emit()`:

```
1 self.valor_actualizado.emit(nuevo_valor)
```

Qt se encarga del resto.

Cómo conectar una señal personalizada

Igual que cualquier otra señal de Qt:

```
1 mi_widget.valor_actualizado.connect(self.actualizar_interfaz)
```

El slot debe recibir los parámetros adecuados.

Ejemplo: Un widget derivado que emite una señal propia

Crear un botón que **cada vez que se pulsa** avisa del nuevo conteo en una etiqueta.

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QMainWindow,
5     QWidget,
6     QVBoxLayout,
7     QLabel,
8     QPushButton
9 )
10 from PySide6.QtCore import Signal
11
12
13 class BotonContador(QPushButton):
14
15     # Señal personalizada que enviará un entero (snake_case)
16     contador_actualizado = Signal(int)
17
18     def __init__(self, parent=None):
19         super().__init__("Pulsado 0 veces", parent)
20         self.__contador = 0
21
22         # Cuando se pulsa el botón estándar de Qt
23         self.clicked.connect(self.__incrementar)
24
25     def __incrementar(self):
26         # Actualizamos el contador interno
27         self.__contador = self.__contador + 1
28
29         # Actualizamos el texto del botón
30         nuevo_texto = "Pulsado " + str(self.__contador) + " veces"
31         self.setText(nuevo_texto)
32
33         # Emitimos la señal con el nuevo valor
34         self.contador_actualizado.emit(self.__contador)
35
36     def contador(self):
```

1. El botón ya tiene la señal `clicked`.
2. Hemos añadido una nueva señal: `contadorCambiado`.
3. Cada vez que cambia el contador:
 - se actualiza el texto,
 - se emite la señal con el nuevo valor.

`clicked` te dice "me han pulsado".

`contadorCambiado` te dice "ya he contado el clic y mi nuevo valor es X", es decir, aporta **un dato que no existía en el widget original**.

Las señales personalizadas sirven para avisar del *estado propio* del widget, no del evento de clic en sí.

Patrón habitual en widgets derivados

1. **Declarar señal** en la clase.
2. **Cambiar estado interno** en un método (p. ej. contador, validación, rango...).
3. **Emitir la señal** cuando el estado cambia.
4. Opcionalmente, **actualizar la interfaz** (texto, colores...).
5. Permitir que otros widgets reaccionen libremente.

Este patrón hace que un widget derivado sea **más profesional, más reutilizable y más predecible**.

3. Crear un componente completamente nuevo

¿Por qué crear un componente propio?

Heredar de `QWidget` es útil cuando queremos diseñar un componente totalmente personalizado.

`QWidget` es la base de todos los widgets de Qt, así que nos da un "lienzo en blanco" para construir nuestro propio control.

Suele utilizarse cuando queremos crear:

- un panel con varios botones y una etiqueta,
- un control que combine distintos widgets pequeños,
- un componente que sirve para agrupar lógica y vista,
- una mini herramienta que queremos reutilizar en varias ventanas.

La idea clave es esta:

Si lo que queremos **no es** un botón, etiqueta, combo o widget predefinido... entonces el punto de partida adecuado es `QWidget`.

Ejemplo: Contador Simple

Vamos a crear un componente llamado `ContadorSimple`.

Este widget tendrá:

- un número mostrado en pantalla,
- un botón "+" para aumentar el valor,
- un botón "-" para disminuirlo.

Aunque internamente contiene tres widgets distintos, a la ventana principal le llegará como **un único componente compacto y reutilizable**.

```
1  from PySide6.QtWidgets import (
2      QApplication,
3      QWidget,
4      QPushButton,
5      QLabel,
6      QHBoxLayout,
7      QVBoxLayout,
8      QMainWindow
9  )
10 from PySide6.QtCore import Qt
11 import sys
12
13
14 class ContadorSimple(QWidget):
15     def __init__(self, parent=None):
16         super().__init__(parent)
17
18         self.__valor = 0
19
20         self.__etiqueta = QLabel("0", self)
21         self.__etiqueta.setAlignment(Qt.AlignCenter)
22
23         self.__boton_mas = QPushButton("+", self)
24         self.__boton_menos = QPushButton("-", self)
25
26         self.__boton_mas.clicked.connect(self.__incrementar)
27         self.__boton_menos.clicked.connect(self.__decrementar)
28
29         layout = QHBoxLayout(self)
30         layout.addWidget(self.__boton_menos)
31         layout.addWidget(self.__etiqueta)
32         layout.addWidget(self.__boton_mas)
33
34     def __incrementar(self):
35         self.__valor = self.__valor + 1
36         self.__etiqueta.setText(str(self.__valor))
```

1. Heredamos de `QWidget`

Esto quiere decir que nuestro componente **no parte del comportamiento de ningún widget concreto.**

Decidimos todo lo que hace.

2. Creamos nuestros propios widgets internos

Nuestro componente contiene tres elementos:

- un botón “-”,
- una etiqueta,
- un botón “+”.

Pero todos están “escondidos” dentro del mismo widget.

3. Definimos un estado privado

`--valor` guarda el número actual.

Nadie desde fuera puede modificarlo directamente (encapsulación).

4. Controlamos los botones desde dentro

Los botones conectan con métodos privados:

- `__incrementar()`
- `__decrementar()`

Ellos actualizan el valor y reflejan el cambio en la etiqueta.

5. Se comporta como una pieza única

Para cualquier ventana que lo utilice, `ContadorSimple` es solo un **widget más**.

No es necesario preocuparse por su funcionamiento interno.

Este patrón es muy habitual cuando queremos crear **controles personalizados reutilizables**.

3.1. `paintEvent` Cuando necesitamos dibujar nosotros

En la mayoría de componentes de Qt, la apariencia ya viene definida: botones, etiquetas, cuadros de texto, listas...

Nosotros los usamos y Qt se encarga de dibujarlos.

Pero hay situaciones donde necesitamos un **componente con un aspecto especial**, algo que no existe tal cual en la biblioteca. Por ejemplo:

- un indicador circular,
- un semáforo,
- un velocímetro,
- un gráfico muy simple,
- un ícono con estilo propio.

En estos casos aparece un método muy importante: `paintEvent`.

¿Qué es `paintEvent`?

`paintEvent` es un método especial de `QWidget`.

Qt lo llama automáticamente cuando el componente tiene que **dibujarse o redibujarse**.

Podemos imaginarlo como si Qt nos dijera:

"Este widget tiene que volver a pintarse."

Si quieres darle una apariencia propia, este es el momento."

Nosotros **no llamamos a `paintEvent`**.

Qt lo hace cuando es necesario.

¿Cuándo llama Qt a `paintEvent`?

Qt repinta un widget cuando:

- aparece por primera vez en pantalla,
- cambia de tamaño,
- ha estado tapado por otro y vuelve a verse,
- otro método cambia su apariencia,
- llamamos a `update()` para pedir que se repinte.

Idea importante:

`update()` no repinta directamente.

Solo avisa a Qt de que debe llamar a `paintEvent` en el siguiente ciclo.

¿Qué hacemos dentro de `paintEvent`?

Dentro de `paintEvent` creamos un `QPainter`, que es el "pincel digital" que Qt nos ofrece para dibujar dentro del widget.

Con `QPainter` podemos dibujar:

- formas (círculos, rectángulos, líneas...),
- colores de fondo,
- bordes,
- texto,
- pequeños gráficos.

Aspectos clave:

- El punto (0,0) está en la esquina superior izquierda del widget.
- El eje X crece hacia la derecha, el Y hacia abajo.
- Muchas funciones usan `self.rect()`, que representa **todo el área visible del widget**.

3.1.1. Dibujos básicos con QPainter

En esta sección aprenderás a dibujar formas y texto usando `QPainter`.

Recuerda: cualquier dibujo debe hacerse dentro de `paintEvent`.

Líneas

Para dibujar una línea entre dos puntos:

```
1 painter.drawLine(x1, y1, x2, y2)
```

Ejemplo:

```
1 painter.drawLine(20, 20, 200, 20)
```

Esto crea una línea horizontal desde `(20, 20)` hasta `(200, 20)`.

Rectángulos

Los rectángulos se dibujan indicando posición y tamaño:

```
1 painter.drawRect(x, y, ancho, alto)
```

- `x` = posición horizontal del **vértice superior izquierdo** del rectángulo.
- `y` = posición vertical de ese mismo vértice.
- `ancho` = tamaño del rectángulo hacia la derecha.
- `alto` = tamaño hacia abajo.

Ejemplo:

```
1 painter.drawRect(20, 50, 150, 80)
```

También existe una versión que acepta un objeto `QRect`:

```
1 rect = QRect(20, 50, 150, 80)
2 painter.drawRect(rect)
```

Esta versión es útil cuando trabajamos con geometrías reutilizables, animaciones o detección de colisiones.

Texto

Para dibujar texto en una posición concreta:

```
1 painter.drawText(x, y, "Texto a mostrar")
```

Ejemplo:

```
1 painter.drawText(30, 150, "Etiqueta de ejemplo")
```

Ten en cuenta que la coordenada **y** marca la **línea base** del texto.

Forma de uso	Ejemplo	¿Para qué sirve?
<code>drawText(x, y, texto)</code>	<code>painter.drawText(30, 150, "Hola")</code>	Texto simple; <code>y</code> marca la línea base.
<code>drawText(QRect, texto)</code>	<code>painter.drawText(rect, "Hola")</code>	Dibuja dentro del rectángulo, alineación por defecto (arriba izquierda).
<code>drawText(QRect, flags, texto)</code>	<code>painter.drawText(rect, Qt.AlignCenter, "Hola")</code>	Control total de alineación: centro, derecha, vertical, etc.
<code>drawText(x, y, w, h, flags, texto)</code>	<code>painter.drawText(20, 40, 100, 50, Qt.AlignRight, "Hola")</code>	Igual que con <code>QRect</code> pero sin crear un objeto.

Círculos y óvalos (`drawEllipse`)

Qt usa elipse como forma general; si ancho y alto son iguales, es un círculo.

```
1 painter.drawEllipse(x, y, ancho, alto)
```

- **x** → posición horizontal del **vértice superior izquierdo** de la caja que contiene la elipse.
- **y** → posición vertical de ese vértice.
- **ancho** → ancho de la caja.
- **alto** → alto de la caja.

Qt dibuja la elipse **encajada** dentro de esa caja.

Ejemplos:

```

1 # Círculo de 80x80
2 painter.drawEllipse(30, 30, 80, 80)
3
4 # Óvalo usando QRect
5 rect = QRect(150, 30, 120, 60)
6 painter.drawEllipse(rect)

```

Puedes usar **QRect** igual que en los rectángulos cuando quieras reutilizar medidas.

3.1.2. Colores y estilos

Qt diferencia entre:

- **Pen** → controla el **borde** (color, grosor).
- **Brush** → controla el **relleno** de las formas.
- **Render Hints** → ajustes que afectan al estilo del dibujo.

Cambiar el color del borde (**setPen**)

Para poner un color simple:

```

1 painter.setPen(QColor(r, g, b))

```

- **r** → componente roja (0-255)
- **g** → componente verde (0-255)
- **b** → componente azul (0-255)

Juntos forman un color siguiendo el modelo **RGB**.

Si además quieres controlar el grosor:

```
1 painter.setPen(QPen(Qt.black, 3))
```

- **color** → puede ser un valor predefinido (`Qt.black`, `Qt.red`...) o un color creado con `QColor(...)`.
- **grosor** → número entero que indica el **espesor** del borde en píxeles.

Este tipo de pen se utiliza en:

- líneas (`drawLine`)
- bordes de rectángulos (`drawRect`)
- bordes de círculos o elipses (`drawEllipse`)

Rellenos de formas (`setBrush`)

El **brush** (pincel de relleno) utiliza los mismos parámetros RGB que `QColor`.

Afecta a:

- Rectángulos
- Elipses y círculos
- Figuras cerradas

Ejemplo:

```
1 painter.setBrush(QColor(180, 180, 250))
2 painter.drawRect(20, 50, 150, 80)
```

Para desactivar el relleno:

```
1 painter.setBrush(Qt.NoBrush)
```

Suavizado de líneas y curvas

(`setRenderHint(QPainter.Antialiasing)`)

El antialiasing mejora la apariencia de:

- Líneas diagonales

- Bordes curvos (círculos, óvalos)
- Borde de letras cuando usamos tamaños grandes

Se activa así:

1

```
painter.setRenderHint(QPainter.Antialiasing)
```

Suele colocarse al inicio de `paintEvent`.

Tabla resumen de métodos

Método	Para qué sirve
<code>setBrush(color)</code>	Define el color o patrón de relleno de formas cerradas.
<code>setPen(QPen)</code>	Cambia el color, grosor y estilo del borde.
<code>setRenderHint(QPainter.Antialiasing)</code>	Suaviza líneas y curvas. Da un acabado más limpio.
<code>drawEllipse(rect)</code>	Dibuja un óvalo o un círculo si ancho = alto.
<code>drawRect(rect)</code>	Dibuja un rectángulo usando un <code>QRect</code> .
<code>drawText(x, y, texto)</code>	Dibuja texto en la posición indicada.

Tabla de importaciones recomendadas para usar en `paintEvent`

Importación	Clase incluida	Para qué se usa
<pre>from PySide6.QtWidgets import QWidget</pre>	QWidget	Clase base para crear widgets personalizados.
<pre>from PySide6.QtGui import QPainter</pre>	QPainter	Objeto que dibuja dentro del widget (figuras, texto, colores...).
<pre>from PySide6.QtGui import QColor</pre>	QColor	Representa colores (RGB, HEX, nombres...).
<pre>from PySide6.QtGui import QPen</pre>	QPen	Define el borde de las formas (color, grosor, estilo).
<pre>from PySide6.QtGui import QBrush</pre>	QBrush	Define el relleno de las figuras (color, degradado).
<pre>from PySide6.QtGui import QFont</pre>	QFont	Configura el tipo de letra para dibujar texto.
<pre>from PySide6.QtCore import QRect</pre>	QRect	Área rectangular para dibujar (coordenadas y tamaño).
<pre>from PySide6.QtCore import QPoint</pre>	QPoint	Coordenadas para dibujar puntos, líneas o figuras.
<pre>from PySide6.QtCore import Qt</pre>	Qt	Constantes útiles: alineaciones, colores básicos, estilos...

3.2. Ejemplos

Ejemplo 1: un círculo rojo

Este es el ejemplo más sencillo para entender cómo funciona `paintEvent`.

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QWidget,
5     QMainWindow,
6     QVBoxLayout
7 )
8 from PySide6.QtGui import QPainter, QColor, QPen
9 from PySide6.QtCore import Qt
10
11
12 class CirculoRojo(QWidget):
13     def paintEvent(self, event):
14         # QPainter es el objeto encargado de dibujar dentro del
15         painter = QPainter(self)
16
17         # Color de relleno del círculo (rojo).
18         painter.setBrush(QColor("red"))
19
20         # Color del borde (negro) usando un QPen, grosor por defecto
21         painter.setPen(QPen(Qt.black))
22
23         # Dibujamos una elipse que ocupa todo el área del widget
24         # Si el widget es cuadrado, la elipse será un círculo perfecto
25         painter.drawEllipse(self.rect())
26
27 class VentanaPrincipal(QMainWindow):
28     def __init__(self):
29         super().__init__()
30
31         self.setWindowTitle("Ejemplo 1: Círculo Rojo")
32         self.resize(300, 300)
33
34         self.setCentralWidget(CirculoRojo())
35
36
```

Qué hace este código:

- Crea un pincel (`QPainter`) para este widget.
- Usa un relleno rojo.
- Dibuja un círculo que ocupa todo el widget.

Nota:

Un widget **solo necesita constructor** cuando guarda información, configura algo inicial o crea subwidgets internos.

Si el widget solo dibuja siempre lo mismo y no tiene estado → **no hace falta escribir `__init__`**.

Ejemplo 2: círculo con borde y texto

Aquí un diseño un poco más elaborado, parecido a un indicador:

```
1 import sys
2 from PySide6.QtWidgets import (
3     QApplication,
4     QWidget,
5     QMainWindow,
6     QVBoxLayout,
7     QPushButton
8 )
9
10 from PySide6.QtGui import QPainter, QColor, QPen
11 from PySide6.QtCore import QRect, Qt
12
13
14 class IndicadorSimple(QWidget):
15     def __init__(self):
16         super().__init__()
17         # Texto que se mostrará dentro del círculo.
18         # Se puede cambiar desde fuera con setTexto().
19         self._texto = "OK"
20
21     def setTexto(self, texto):
22         # Guardamos el nuevo texto.
23         self._texto = texto
24         # update() avisa a Qt de que debe volver a dibujar el widget.
25         self.update()
26
27     def paintEvent(self, event):
28         # QPainter es el objeto que permite dibujar dentro del widget.
29         painter = QPainter(self)
30
31         # Activamos el suavizado de bordes para evitar formas "cuadradas".
32         painter.setRenderHint(QPainter.Antialiasing)
33
34         # Configuramos el color de relleno del círculo (verde).
35         painter.setBrush(QColor("#4CAF50"))
36
37         # Borde del círculo en color negro.
```

Nota: Cálculo de X (horizontal)

```
(self.width() - lado) // 2
```

Significa:

1. "¿Cuánto espacio sobra a los lados?"

→ `self.width() - lado`

2. "Reparte ese espacio: mitad para izquierda y mitad para derecha."

→ `// 2`

El resultado es la **coordenada X del vértice superior izquierdo** del cuadrado.