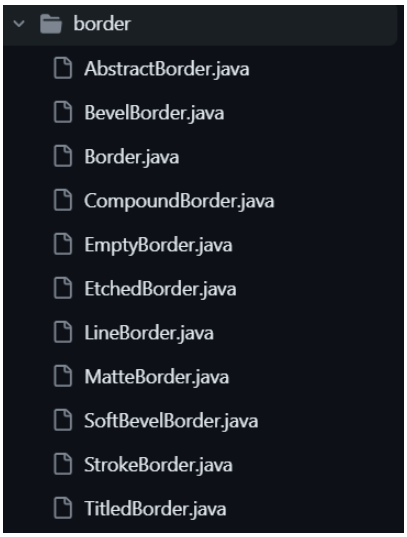
# Taller 5

- **Información general del proyecto:**

  Para este taller se escogió Java Swing, el cual se puede acceder a través del repositor openjdk:

  https://github.com/openjdk/jdk/tree/master/src/java.desktop/share/classes/javax/swing

  La biblioteca Swing de Java es un conjunto de clases y componentes gráficos que se utiliza para crear interfaces de usuario (UI) en aplicaciones de escritorio. Esta es bastante robusta y tiene demasiadas clases para analizarlas todas como un solo proyecto, por lo que se escogió un solo directorio dentro de la librería que contiene 11 clases, este se llama border:

  En general, la estructura del diseño de Swing se parece mucho al modelo MVC (Model-View-Controller). Este directorio (border) al igual que otros dentro de la librería, solo representan la lógica y operaciones de la aplicación por lo que estas clases por si solas no pueden visualizarse (son el Modelo de la libreria), para eso se usan otras clases como JFrame, que extiende Frame de AWT, la cual a su vez extiende Window, que si se puede visualizar, y haría parte de las clases del View.

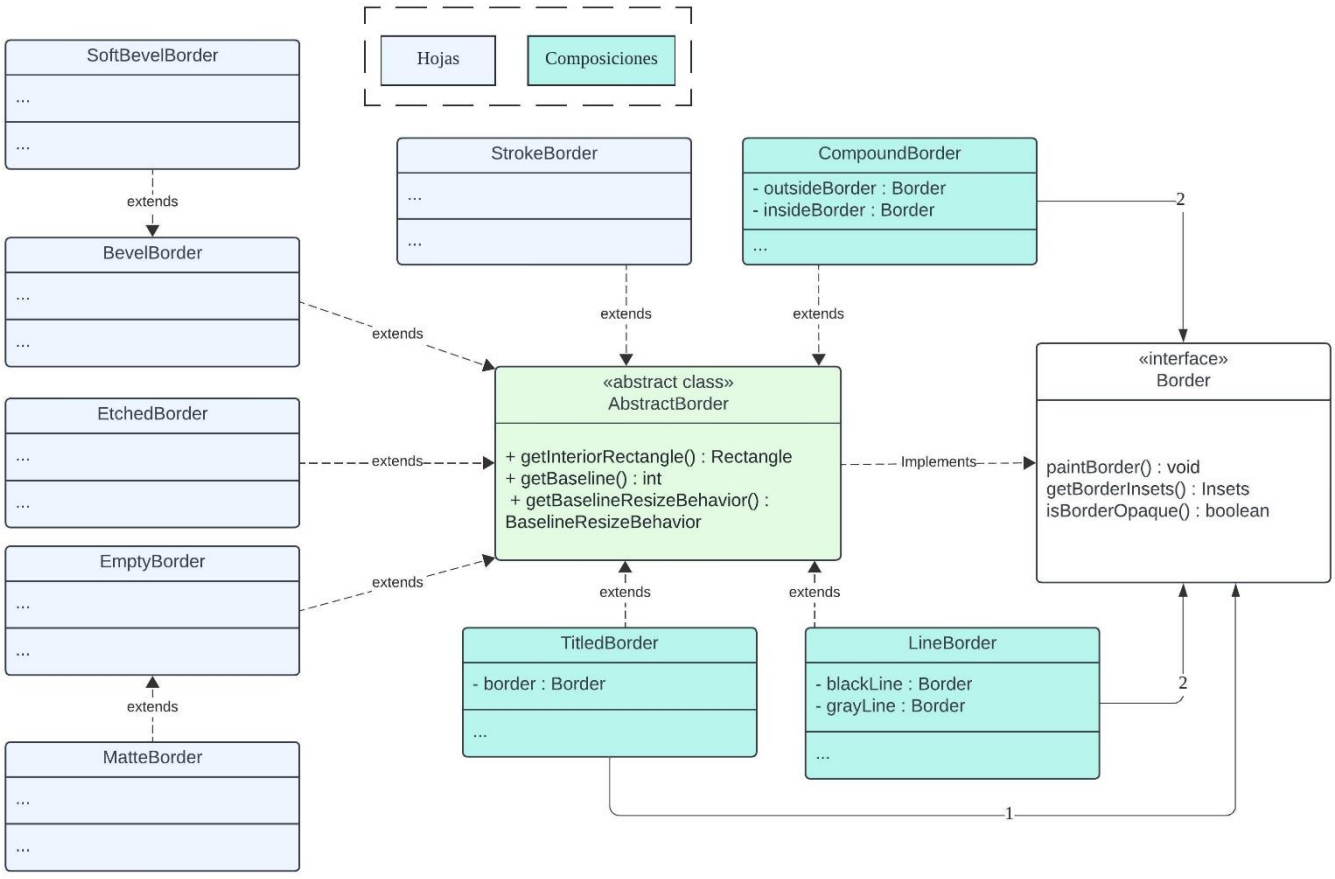
  Los bordes en Swing son elementos visuales que rodean los componentes gráficos, como botones, paneles y cuadros de texto.

- **Patrón**: **Composite**

Siguiendo la definición del repositorio java-design-patterns, el patrón Composite sirve para crear objetos en estructuras de árbol para representar jerarquías de partes y totalidades. Esto permite a los clientes manejar objetos individuales y composiciones de objetos de manera uniforme.

Hablando del proyecto en cuestión, Intuitivamente podemos pensar que existen varios tipos de bordes, de diferentes tamaños, grosores, y colores; y algunos serán parecidos a otros, por lo que, idealmente sería una buena solución reutilizar código si un borde tiene muchas cosas en común con otro, y estandarizar un comportamiento para todos. Así, se crearía un árbol de objetos, donde la raíz seria la interfaz principal 'Border.java', y el resto de clases serian sus hojas y composiciones de bordes.

Aquí se muestra un diagrama UML simplificado de la Aparición del patrón en el proyecto




En el diagrama podemos encontrar dos cosas:

**Hojas u objetos individuales:** Se llaman hojas porque no están compuestos por más bordes, son objetos individuales (un borde individual), pero sí implementan al igual que todos los elementos del árbol, la interfaz Border.java

**Composiciones:** Son bordes que están compuestos de otros bordes, por lo que en sus atributos podemos encontrar 'blackLine' o 'insideBorder', son bordes compuestos.

- **Ventajas**

  La ventaja de usar este patrón es que podemos llamar los mismos métodos de una composición y una hoja, ya que queremos que se comporten de la misma manera, uniformemente. Desde afuera un LineBorder (composición) se comporta igual que un EtchedBorder (hoja), porque ambos se pueden declarar como una variable Border:

```
Border lineBorder = new LineBorder(Color.BLACK);
Border etchedBorder = new EtchedBorder();
```

  Note que también tenemos dos clases 'SoftBevelBorder' y 'MatteBorder' que son las únicas que no extienden la clase abstracta principal 'AbstractBorder', ya que, al ser una modificación de otro borde, se puede reutilizar código, gracias a la estructura raíz-hoja.

- **Posibles desventajas**

  Una posible desventaja puede ser las restricciones en operaciones, ya que puede darse el caso que no todos los métodos y operaciones puedan ser aplicados de manera uniforme a todos los elementos del árbol, pues quizás unos bordes difieran mucho de otros. Es decir, si quisiéramos hacer una borde muy especial, con características muy exóticas, puede que conflictúe con el comportamiento de la interfaz Border.java, y en dado caso tendríamos que modificar la interfaz o usar otro Patrón llamado Adapter, el cual podría ser de utilidad, pero haría más complicado el código.

- **Patrones alternativos**

  Se podría utilizar Decorator, tendríamos que primero hacer un borde base, y luego hacer varias versiones de este 'decorando' al inicial para obtener otros tipos de borde. El reto estaría en definir como seria el 'borde base', ya que todos los bordes nacerían de agregarle cosas a este, y quizás no es posible hacer un borde base.

  También se podría utilizar Factory, sin embargo, ya no tendríamos una clase para cada tipo de borde, sino habría una sola clase 'BorderFactory' que nos crearía instancias de bordes personalizados, así el constructor podría tener varios parámetros como emptyBorder = False, compoundBorder = null, etc. El problema quizás seria que los bordes tienen tantas características que el constructor sería muy grande y la clase BorderFactory muy saturada.