

Universidad de Las Américas
Facultad de Ingenierías y Ciencias Agropecuarias
Ingeniería de Software
Informe de laboratorio

1. DATOS DEL ALUMNO: Camila Cabrera

2. TEMA DE LA PRÁCTICA: Implementación Abstract Factory

3. OBJETIVO DE LA PRÁCTICA

- Realizar el ejercicio propuesto de implementación de Abstract Factory para mejorar la comprensión de su aplicación.

4. OBJETIVOS ESPECÍFICOS

- Leer y comprender los principios y características del patrón Abstract Factory.
- Inicializar Visual Studio Code para proyectos de java.
- Crear un proyecto de java en visual Studio code.
- Implementar la documentación para el proyecto ejemplo del patrón Abstract Factory.

5. MATERIALES Y MÉTODOS

Materiales:

- Visual studio code
- JDK 17
- Guía práctica de Patrón Abstract Factory

Métodos:

1. Validación del JDK 17.
2. Configuración de Visual Studio Code para proyectos de Java.
3. Creación del proyecto Java sin herramientas de construcción del proyecto.

4. Implementación de la guía práctica del Abstract Factory.
5. Registro de hallazgos durante el proceso de implementación de la documentación.

6. DESARROLLO DE LA PRÁCTICA Y RESULTADOS

Marco Teórico

“Abstract Factory es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.” (Refactoring Guru, s.f.). Permite crear objetos que siguen cierta estructura común, pueden tener implementaciones específicas según el tipo de problema. Provee una serie de reglas o instrucciones que permiten crear diferentes tipos de objetos sin saber exactamente cuál es. Este patrón se considera similar al método de Fabrica, pero se considera como más abstracto y engloba un mayor aspecto. En este se trabaja con un patrón que implementa una super factory que crea factories.

Abstract Factory declara un conjunto de métodos, cada uno de los cuales es responsable de crear un tipo específico de objeto y garantiza que las fábricas concretas estén vinculadas a una interfaz común, proporcionando una forma consistente de crear conjuntos de objetos relacionados.

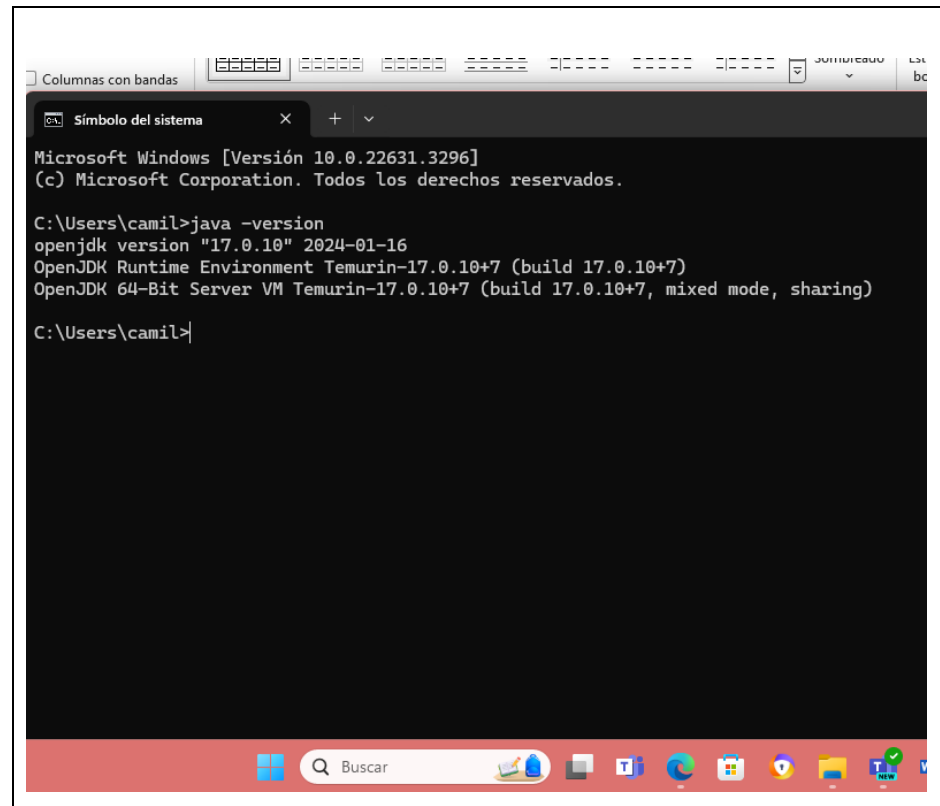
Los productos abstractos representan una familia de objetos relacionados definiendo un conjunto de métodos o propiedades comunes. Actúa como un tipo abstracto o de interfaz al que deben adherirse todos los productos concretos dentro de una familia, proporcionando una forma coherente para que los productos concretos se utilicen indistintamente.

Se lo aplica para tener la certeza de que los objetos creados son compatibles entre ellos, lo que genera un acoplamiento fuerte entre el producto y el código del cliente. Sin embargo, se debe manejarlo de una manera eficiente porque el código puede llegar a complicarse al manejar las interfaces.

Desarrollo de la práctica

- **Validación del JDK 17:**
 - Para verificar la versión del JDK se comprueba con el comando “java -version” en la terminal del computador.

Imagen #1: Validación versión JDK



```
Microsoft Windows [Versión 10.0.22631.3296]
(c) Microsoft Corporation. Todos los derechos reservados.

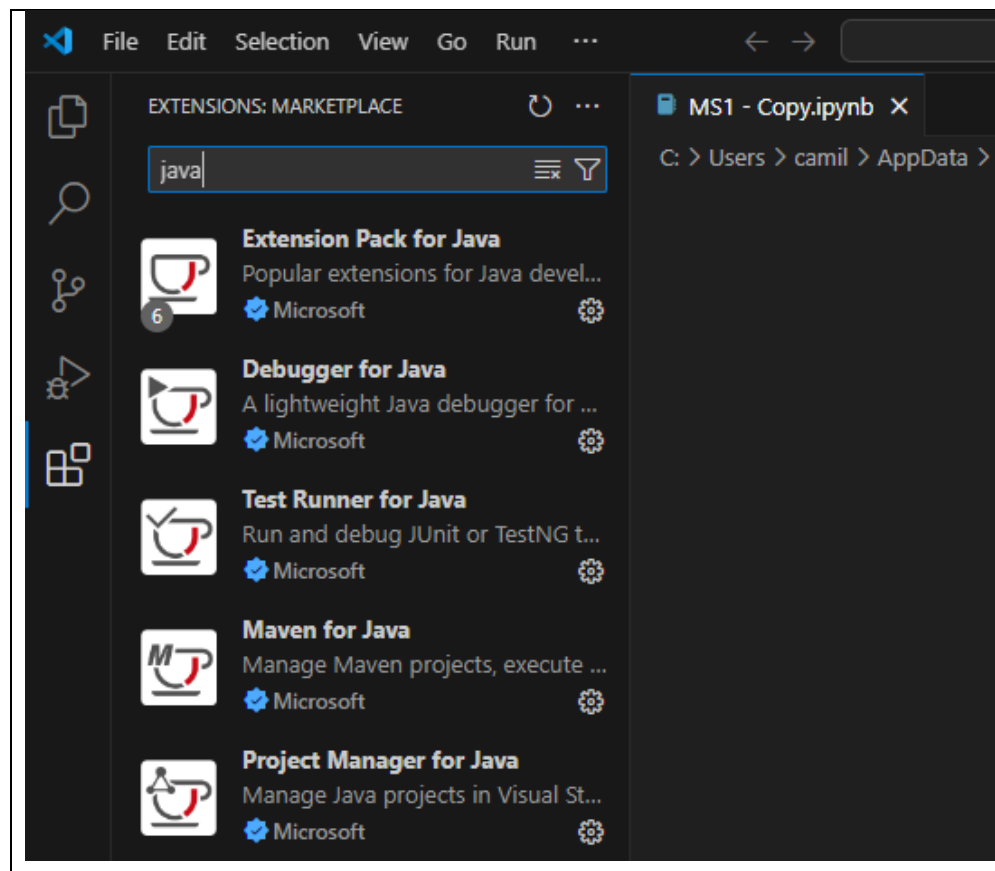
C:\Users\camil>java -version
openjdk version "17.0.10" 2024-01-16
OpenJDK Runtime Environment Temurin-17.0.10+7 (build 17.0.10+7)
OpenJDK 64-Bit Server VM Temurin-17.0.10+7 (build 17.0.10+7, mixed mode, sharing)

C:\Users\camil>
```

Fuente: Elaboración propia

- **2. Configuración de Visual Studio Code para proyectos de Java.**
 - En el IDE visual studio code se debe instalar o validar la dependencia para Java (Imagen 2) .

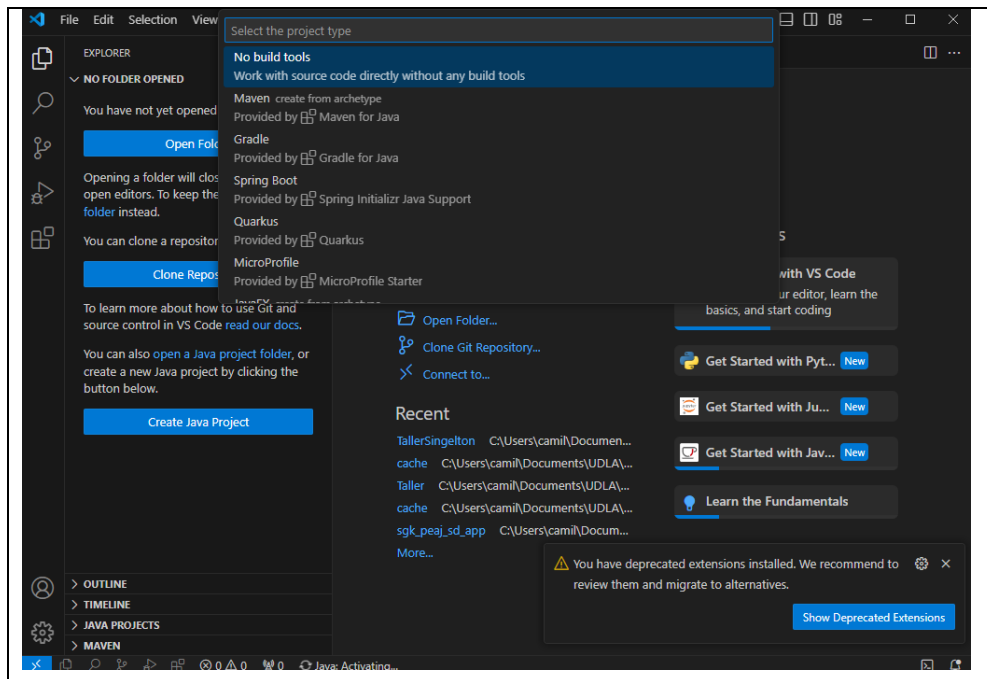
Imagen #2: Dependencia de Java



Fuente: Elaboración propia

- **Creación del proyecto Java sin herramientas de construcción del proyecto.**
 - Se inicializo el proyecto de Java escogiendo la opción sin herramientas de construcción.

Imagen #3: Dependencia de Java



- **Implementación de la guía práctica del patrón Singleton.**
 - Se siguió la guía práctica de Patron Singleton obtenida en el siguiente link: [Builder en Java / Patrones de diseño \(refactoring.guru\)](https://refactoring.guru/design-patterns/builder-in-java) para la elaboración del código.
 - La práctica consistió en implementar el método Abstract Factory en un proyecto en java.

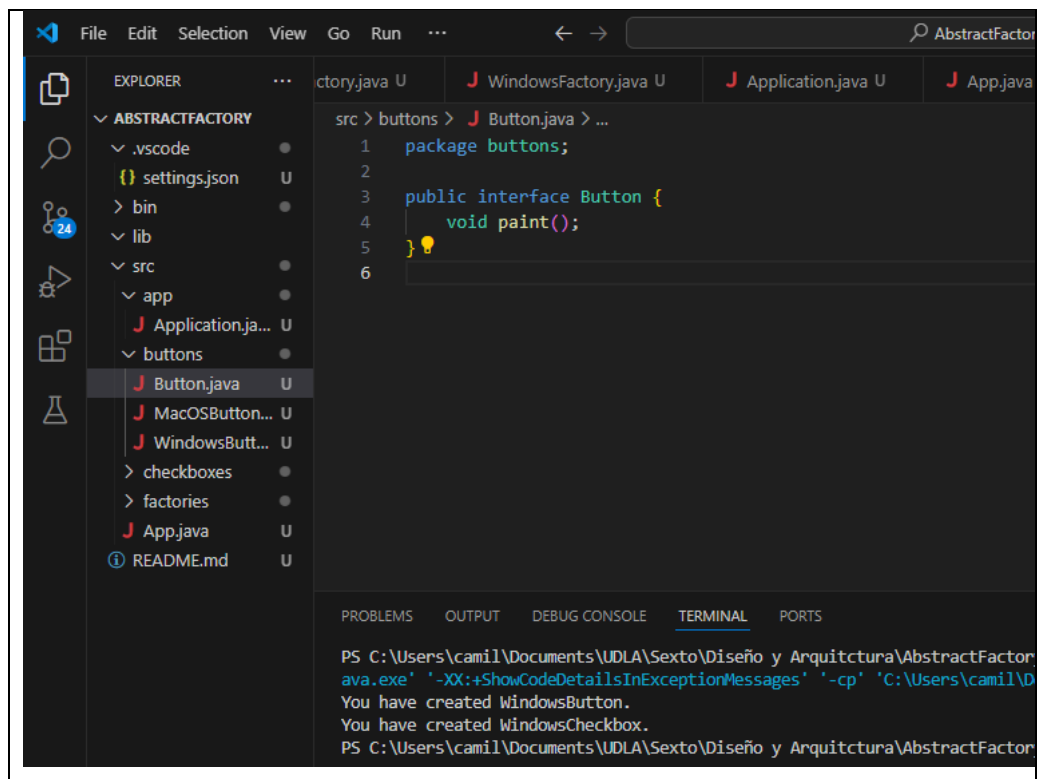
ABSTRACT FACTORY

Carpeta: buttons

Interfaz: Button

Declara el método void Paint.

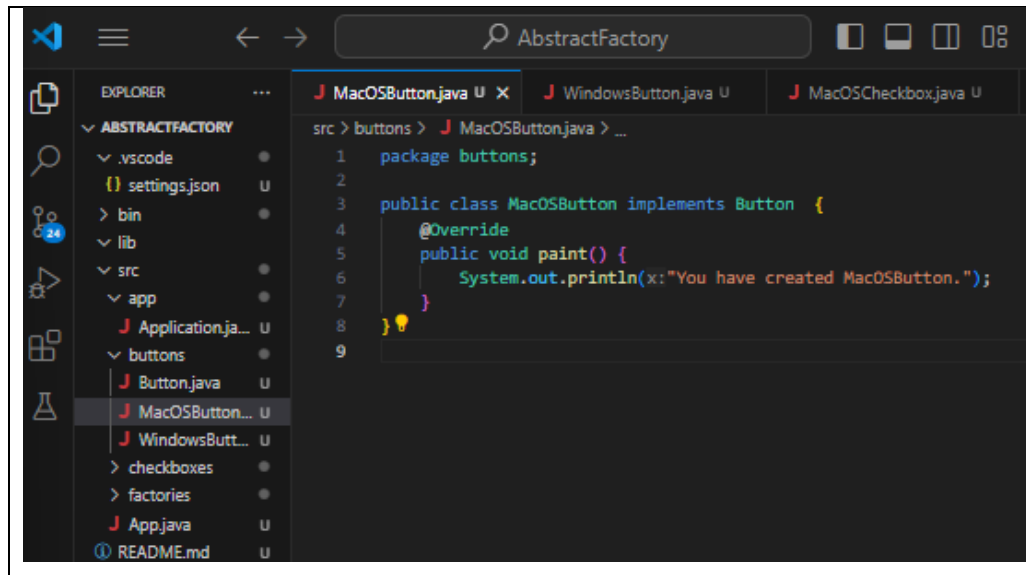
Imagen #4: Interfaz Button



Clase: MacOSButton

Implementa la interfaz Button y desarrolla el método void Paint, con la impresión de "You have created MacOSButton."

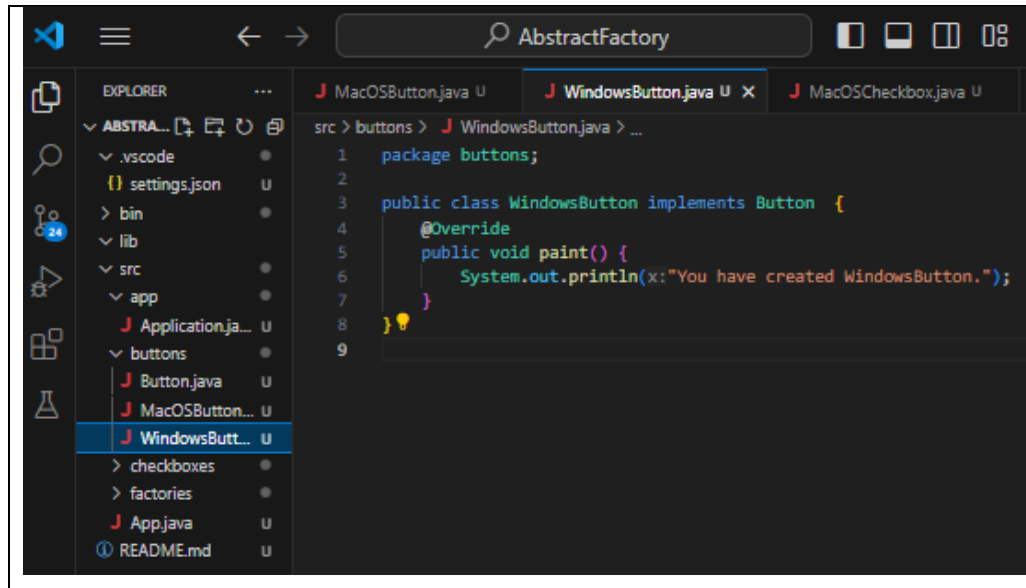
Imagen #5: Clase MacOSButton



Clase: WindowsButton

Implementa la interfaz Button y desarrolla el método void Paint, con la impresión de " You have created WindowsButton".

Imagen #6: Clase WindowsButton

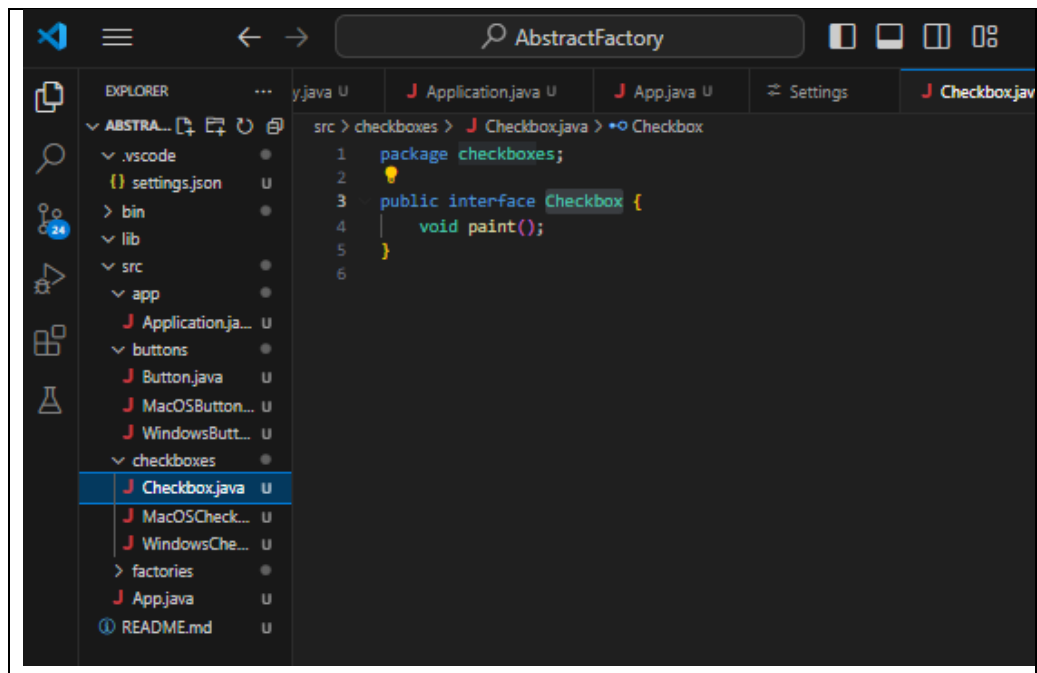


Carpeta: cheackboxes

Interfaz: Checkbox

Declara el método void Paint.

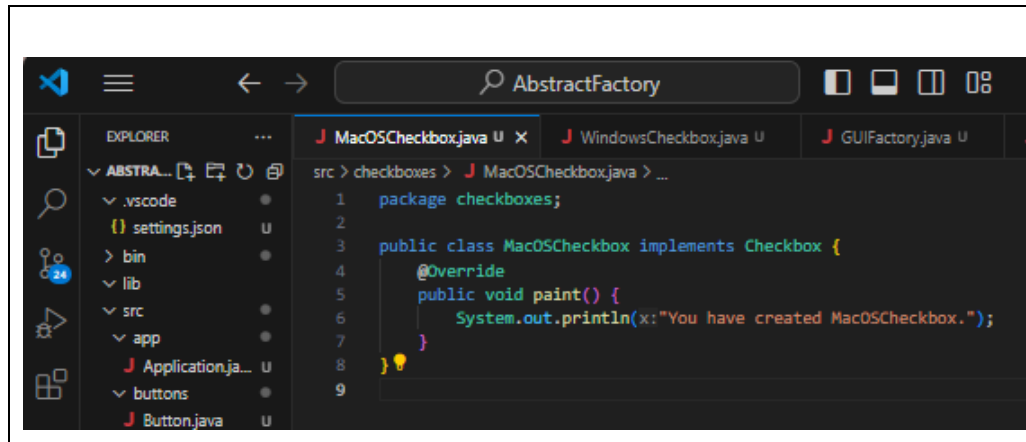
Imagen #7: Interfaz Checkbox



Clase: MacOSCheckbox

Implementa la interfaz Checkbox y desarrolla el método void Paint, con la impresión de " You have created WindowsButton".

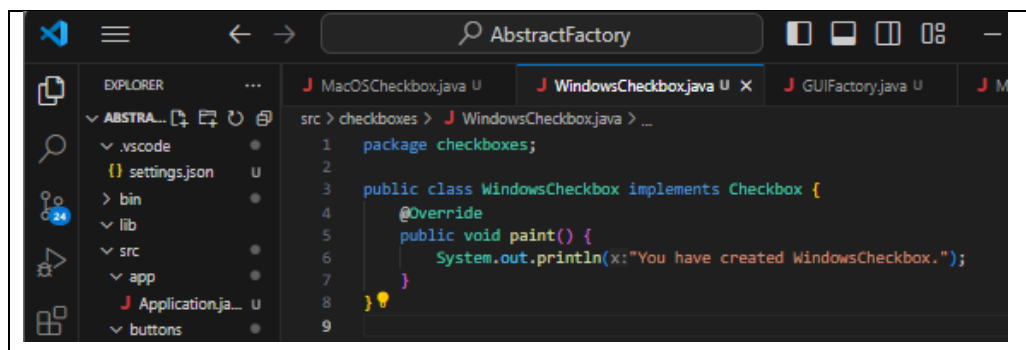
Imagen #8: Clase MacOSCheckbox



Clase: WindowsCheckbox

Implementa la interfaz Checkbox y desarrolla el método void Paint, con la impresión de " You have created WindowsCheckbox.".

Imagen #9: Clase WindowsCheckbox

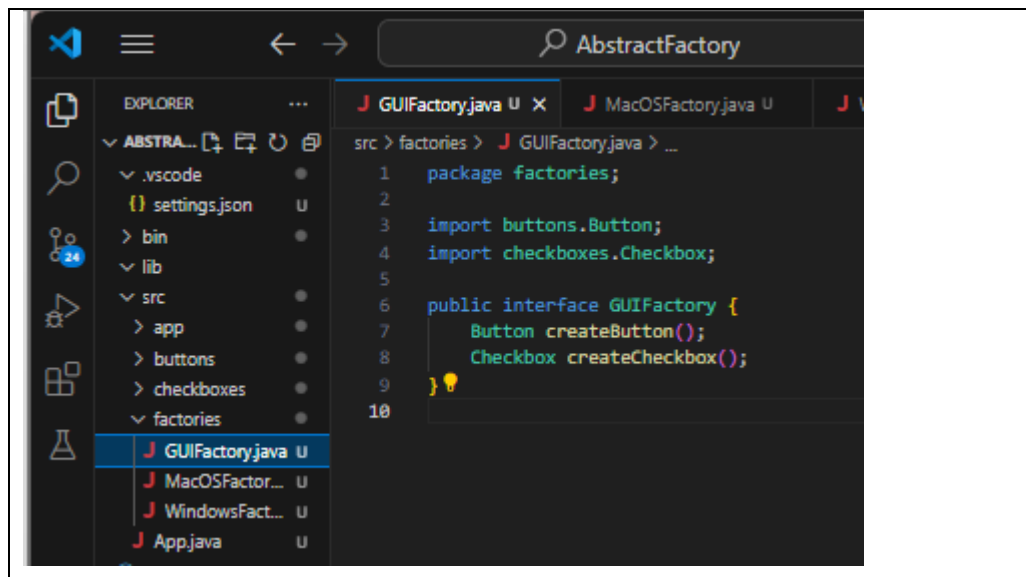


Carpeta: factories

Clase: GUIFactory

Declara dos métodos, el primero retorna un tipo Button y el segundo un tipo Checkbox, haciendo referencia a las interfaces.

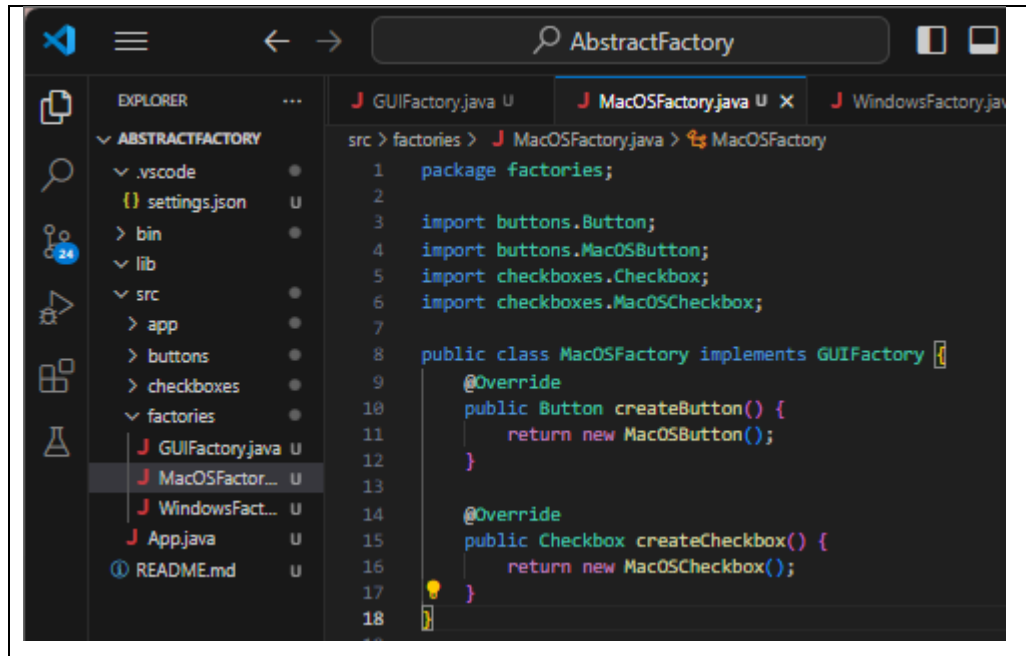
Imagen #10: Interfaz GUIFactory



Clase: MacOSFactory

Implementa la interfaz GUIFactory y desarrolla los métodos Button createButton y Checkbox MacOSCheckbox(), estos retornan las clases previamente creadas referentes a Windows.

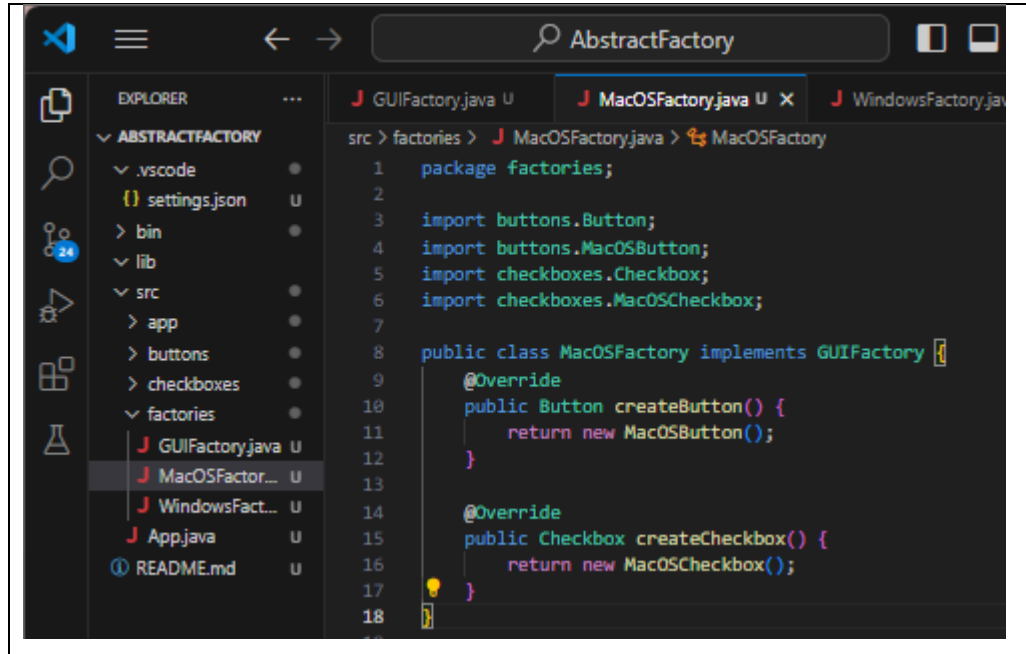
Imagen #11: Clase MacOSFactory



Clase: WindowsFactory

Implementa la interfaz GUIFactory y desarrolla los métodos Button createButton y Checkbox createCheckbox (), estos retornan las clases previamente creadas referentes a MacOS.

Imagen #12: Clase MacOSFactory

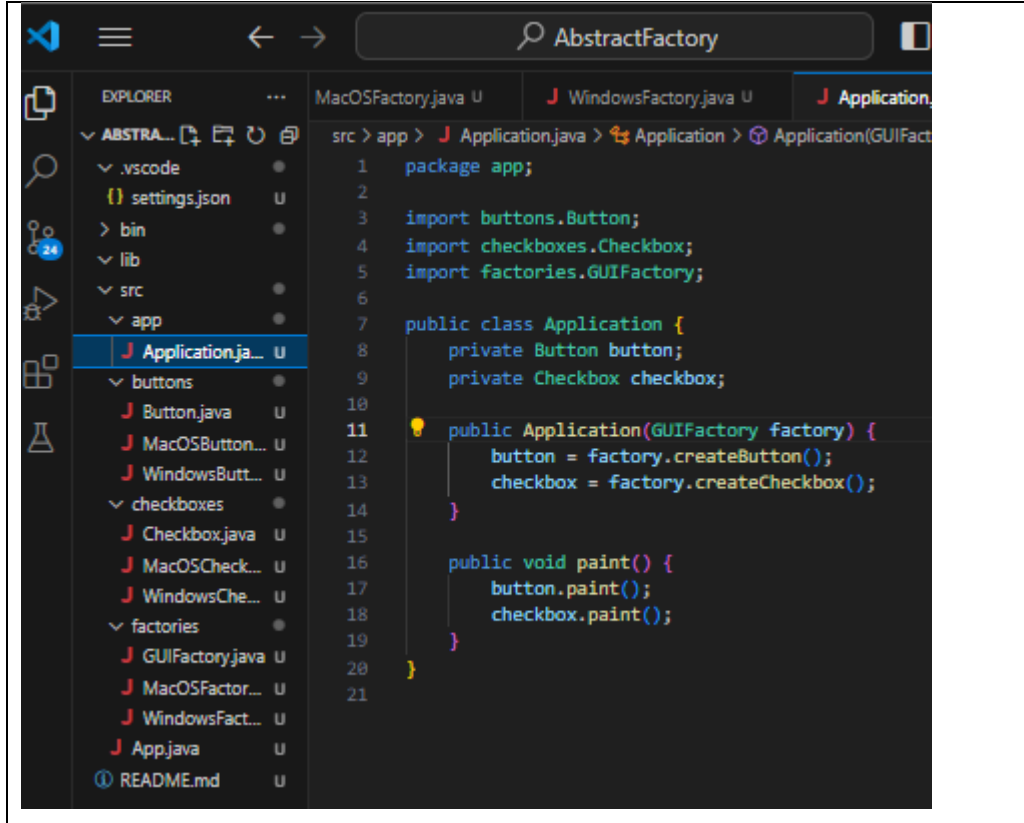


Carpeta: App

Clase: Application

Es una clase que tiene dos atributos privados que almacenan objetos de la clase Button y Checkbox. En su constructor recibe un objeto de tipo GUIFactory. En esta clase hay un método Paint que llama a los métodos Paint de cada objeto.

Imagen #13: Clase Application

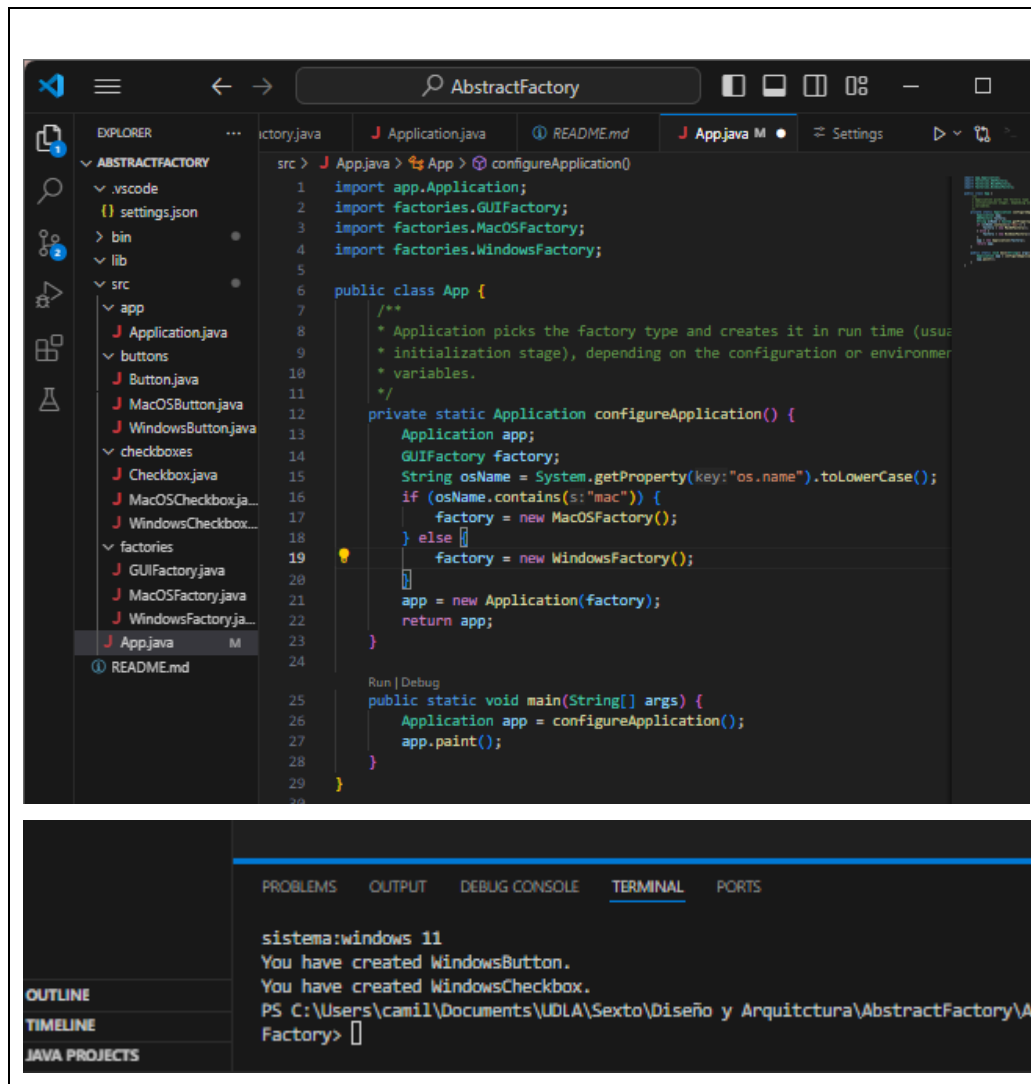


```
src > app > Application.java > Application > Application(GUIFact
1  package app;
2
3  import buttons.Button;
4  import checkboxes.Checkbox;
5  import factories.GUIFactory;
6
7  public class Application {
8      private Button button;
9      private Checkbox checkbox;
10
11     public Application(GUIFactory factory) {
12         button = factory.createButton();
13         checkbox = factory.createCheckbox();
14     }
15
16     public void paint() {
17         button.paint();
18         checkbox.paint();
19     }
20 }
21
```

Clase: App (Main)

Dentro del método `configureApplication()`, se crea una instancia de `Application` y una instancia de `GUIFactory`. La instancia de `GUIFactory` será de tipo `MacOSFactory` si el sistema operativo es `MacOS`, de lo contrario será de tipo `WindowsFactory`. Una vez que se ha seleccionado la fábrica adecuada según el sistema operativo, se crea una instancia de `Application` pasándole la fábrica correspondiente como argumento. En este caso se usa la fábrica de `Windows` ya que se está corriendo en `Windows 11`.

Imagen #14: App



- **Pruebas y Evaluación de los Resultados:**
 - Como resultado se pudo cumplir con la guía de la práctica y comprender cómo se puede utilizar el patrón Abstract Factory para crear objetos que puedan funcionar en diferente contexto dado que comparten la misma fabrica que los genera.

7. OPINIÓN PERSONAL

- En clase habíamos explorado el método Factory que implementa una fábrica para poder crear varios objetos cuando no sabes en tiempo de ejecución cual deberá ser el seleccionado. En base a la investigación previa a la práctica menciona que la relación del Abstract factory con el patrón factory es su mayor abstracción. Al realizar esta práctica, me quedo más claro esta idea ya que básicamente este patrón es una fábrica de fábricas, que crear familias de objetos relacionados sin especificar sus clases concretas. Esto es crucial para el desarrollo de software modular y flexible.
- Considero que implementar este patrón en escenarios de prueba, me hacen comprender como poder desarrollarlos y en que entorno se puede aplicar. Considero que se debe comprender cómo y cuándo aplicarlo para poder hacer que el código sea robusto y eficiente. Por ejemplo, si no se identifica correctamente las familias de objetos y los ejes de cada uno como sistema operativo y tipo de objeto de UI (botón, checkbox) puede ser que el patrón se implemente incorrectamente. Por tanto, lo primordial en este patrón es el análisis de relaciones.
- Me parece importante ver y entender como cuando ya está implementado este patrón se puede introducir fácilmente más elementos que se relacionen a las familias de objetos, por ejemplo, se podría agregar un tipo de objeto que sea message, el cual va a existir tanto para el S.O Windows y Mac, sin necesidad de regenerar la estructura del código por completo.
- Esta práctica me hizo analizar en los campos en los que actualmente podría aplicarlo, llegue a la conclusión que podría usarlo en el desarrollo de videojuegos. Ya que cada fábrica abstracta puede ser la responsable de crear una familia de objetos relacionados como enemigos, otra que sea escenas, otra de tipo ganancias o recompensa, etc. Por lo que puedo decir que aprender un patrón es relacionarlo con su aplicación en contextos reales, y de esta manera se pude tener más claro el panorama.

8. ANEXOS

Link Github del proyecto: [AbstractFactory/src at master · CamilaACT/AbstractFactory](https://github.com/CamilaACT/AbstractFactory)
(github.com)

9. BIBLIOGRAFÍA

Refactoring Guru. (s.f.). *Refactoring Guru*. Obtenido de Abstract Factory:
<https://refactoring.guru/es/design-patterns/abstract-factory>