7 Creational Design Pattern: Assignment — Abstract Factory Pattern for a Cross-Platform GUI Toolkit

**Concept** The Abstract Factory pattern lets a client create families of related objects (buttons, check-boxes, menus, etc.) through a common interface, keeping client code independent of platform-specific classes and ensuring that “matching” components are always used together.

#### **Starter code (keep unchanged in src/main/java/legacy/)**

package legacy;

public interface Button { void render(); }

package legacy;

public interface Checkbox { void render(); }

package legacy;

public class WindowsButton implements Button {

public void render(){ System.out.println("Legacy Windows button"); }

}

package legacy;

public class MacOSButton implements Button {

public void render(){ System.out.println("Legacy MacOS button"); }

}

package legacy;

public class WindowsCheckbox implements Checkbox {

public void render(){ System.out.println("Legacy Windows checkbox"); }

}

package legacy;

public class MacOSCheckbox implements Checkbox {

public void render(){ System.out.println("Legacy MacOS checkbox"); }

}

package legacy;

public class Client {

public static void main(String[] args){

Button btn = new WindowsButton(); // direct dependency

Checkbox cb = new WindowsCheckbox(); // direct dependency

btn.render(); cb.render();

}

}

#### **Tasks**

1 Explain in analysis/abstract\_factory\_problems.md why the client is tightly coupled to concrete GUI classes and list two risks that follow.  
 2 Create abstraction GUIFactory with methods createButton() and createCheckbox().  
 3 Provide concrete factories WindowsFactory, MacOSFactory.  
 4 Write a new client class Application that receives a GUIFactory through its constructor, creates the widgets, and renders them. The old Client remains untouched in the legacy package.  
 5 Add support for a new platform Linux by creating LinuxButton, LinuxCheckbox, and LinuxFactory—show that Application runs without modification.  
 6 Reflection (reflection.md)  
 • How did Abstract Factory enforce family consistency?  
 • Which architecture pattern did you apply and how does it compare with the Simple Factory?

#### **Deliverables**

analysis/abstract\_factory\_problems.md

src/main/java/clean/gui/\*\* ← abstract + concrete products

src/main/java/clean/factory/\*\* ← GUIFactory + platform factories

src/main/java/clean/Application.java

src/main/java/clean/Main.java ← selects factory at runtime

reflection.md

README.md

#### **Solution reference**

// clean/gui/Button.java

package clean.gui;

public interface Button { void render(); }

// clean/gui/Checkbox.java

package clean.gui;

public interface Checkbox { void render(); }

// clean/gui/WindowsButton.java

package clean.gui;

public class WindowsButton implements Button {

public void render(){ System.out.println("Rendering Windows button."); }

}

// clean/gui/MacOSButton.java

package clean.gui;

public class MacOSButton implements Button {

public void render(){ System.out.println("Rendering MacOS button."); }

}

// clean/gui/LinuxButton.java

package clean.gui;

public class LinuxButton implements Button {

public void render(){ System.out.println("Rendering Linux button."); }

}

// clean/gui/WindowsCheckbox.java

package clean.gui;

public class WindowsCheckbox implements Checkbox {

public void render(){ System.out.println("Rendering Windows checkbox."); }

}

// clean/gui/MacOSCheckbox.java

package clean.gui;

public class MacOSCheckbox implements Checkbox {

public void render(){ System.out.println("Rendering MacOS checkbox."); }

}

// clean/gui/LinuxCheckbox.java

package clean.gui;

public class LinuxCheckbox implements Checkbox {

public void render(){ System.out.println("Rendering Linux checkbox."); }

}

// clean/factory/GUIFactory.java

package clean.factory;

import clean.gui.Button;

import clean.gui.Checkbox;

public interface GUIFactory {

Button createButton();

Checkbox createCheckbox();

}

// clean/factory/WindowsFactory.java

package clean.factory;

import clean.gui.\*;

public class WindowsFactory implements GUIFactory {

public Button createButton(){ return new WindowsButton(); }

public Checkbox createCheckbox(){ return new WindowsCheckbox(); }

}

// clean/factory/MacOSFactory.java

package clean.factory;

import clean.gui.\*;

public class MacOSFactory implements GUIFactory {

public Button createButton(){ return new MacOSButton(); }

public Checkbox createCheckbox(){ return new MacOSCheckbox(); }

}

// clean/factory/LinuxFactory.java

package clean.factory;

import clean.gui.\*;

public class LinuxFactory implements GUIFactory {

public Button createButton(){ return new LinuxButton(); }

public Checkbox createCheckbox(){ return new LinuxCheckbox(); }

}

// clean/Application.java

package clean;

import clean.factory.GUIFactory;

import clean.gui.Button;

import clean.gui.Checkbox;

public class Application {

private final Button button;

private final Checkbox checkbox;

public Application(GUIFactory factory){

this.button = factory.createButton();

this.checkbox = factory.createCheckbox();

}

public void renderUI(){

button.render();

checkbox.render();

}

}

// clean/Main.java

package clean;

import clean.factory.\*;

public class Main {

public static void main(String[] args){

String os = System.getProperty("os.name"); // simplistic detection

GUIFactory factory = os.startsWith("Windows") ? new WindowsFactory()

: os.startsWith("Mac") ? new MacOSFactory()

: new LinuxFactory();

Application app = new Application(factory);

app.renderUI();

}

}

**Pattern used** Abstract Factory — a “factory of factories” returning families of related components that work together.