6 Creational Design Pattern: Assignment — Simple Factory Pattern in a Logger Module

**Objective** Demonstrate that you can centralise object-creation logic with a factory, extend functionality without touching client code, and articulate how the Simple Factory relates to other creational patterns.  
 **Concept recap** A Simple Factory delegates instantiation to one class; clients depend only on an abstraction (interface) while the factory decides which concrete type to return. This reduces coupling and keeps creation logic in one place.

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

package legacy;

public interface Logger { void log(String message); }

package legacy;

public class ConsoleLogger implements Logger {

public void log(String msg){ System.out.println("Console: "+msg); }

}

package legacy;

public class FileLogger implements Logger {

public void log(String msg){ System.out.println("File: "+msg); }

}

package legacy;

public class App {

public static void main(String[] args){

Logger console = new ConsoleLogger(); // client knows concrete type

console.log("legacy console");

Logger file = new FileLogger(); // client again instantiates

file.log("legacy file");

}

}

#### **Tasks**

1 Add a new requirement: support a DatabaseLogger without modifying existing client code. Explain in analysis/factory\_problems.md why direct instantiation violates Open/Closed and increases coupling.  
 2 Implement a **Simple Factory** (LoggerFactory) in src/main/java/clean/ that chooses the logger by a string key.  
 3 Refactor the App program so it requests loggers exclusively through the factory. Existing logger classes must remain unchanged.  
 4 Prove extensibility: add DatabaseLogger and obtain it via the factory with no edits to App.java.  
 5 Reflection (reflection.md)  
 • How did the Simple Factory decouple creation from usage?  
 • Which architecture pattern is this and how might it evolve into the Factory Method or Abstract Factory?  
 • What trade-offs come with a string-based factory (runtime errors, enum alternatives)?

#### **Deliverables**

analysis/factory\_problems.md

src/main/java/clean/LoggerFactory.java

src/main/java/clean/DatabaseLogger.java

src/main/java/clean/App.java ← refactored client

reflection.md

README.md ← compile/run instructions

#### **Solution reference**

package clean;

public interface Logger { void log(String message); }

package clean;

public class ConsoleLogger implements Logger {

public void log(String m){ System.out.println("Console Logger: "+m); }

}

package clean;

public class FileLogger implements Logger {

public void log(String m){ System.out.println("File Logger: "+m); }

}

package clean;

public class DatabaseLogger implements Logger {

public void log(String m){ System.out.println("DB Logger: "+m); }

}

package clean;

public final class LoggerFactory {

public static Logger createLogger(String type){

switch(type.toLowerCase()){

case "console": return new ConsoleLogger();

case "file": return new FileLogger();

case "db": return new DatabaseLogger();

default: throw new IllegalArgumentException("Unknown logger: "+type);

}

}

}

package clean;

public class App {

public static void main(String[] args){

Logger console = LoggerFactory.createLogger("console");

console.log("hello console");

Logger file = LoggerFactory.createLogger("file");

file.log("hello file");

Logger db = LoggerFactory.createLogger("db");

db.log("hello database");

}

}

**Pattern used** Simple Factory (sometimes called Static Factory); can later evolve to Factory Method (factory hierarchy) or Abstract Factory (families of related objects).