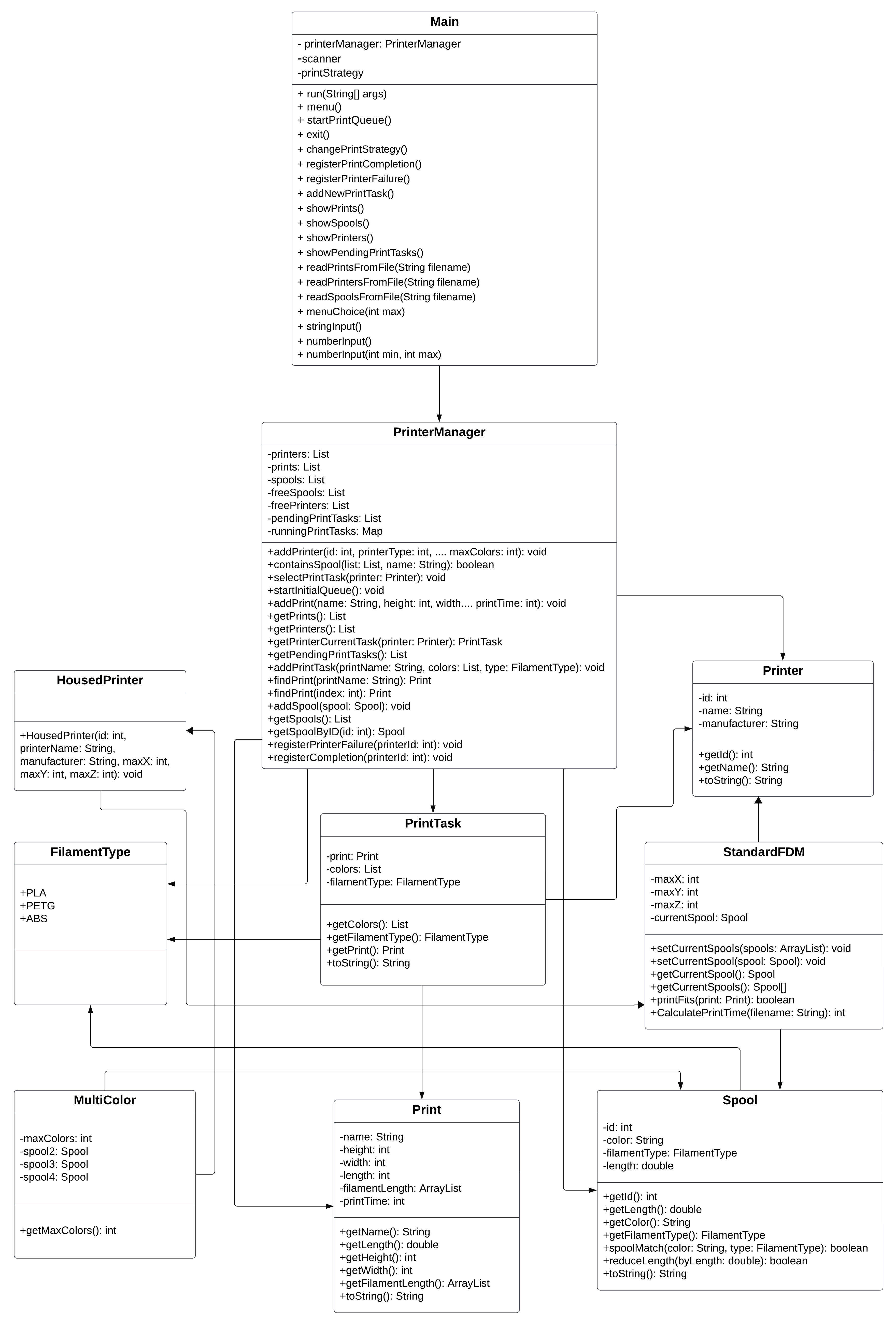
Technical Report

1. Code Analysis

1.1 Class Diagram



1.2 Code Smells

Several code smells have been identified when analyzing the existing code of the application.

Following are the details on each of them:

1) Code Smell: Bloaters (Large Class)

– Where: Main.java

– Whats wrong: Main class contains too many lines of code (fields/methods).

– Why is it wrong: The Main class code has grown to such huge proportions that it is difficult to work with.

– Possible solution: Extraction. This includes dividing a part of a Main class behavior into separate components.

2) Code Smell: Bloaters (Large Class)

– Where: PrinterManager.java

– Whats wrong: PrinterManager class contains too many lines of code (fields/methods).

– Why is it wrong: The PrinterManager class code has grown to such huge proportions that it is difficult to work with.

– Possible solution: Extraction. This includes dividing a part of a PrinterManager class behavior into separate components.

3) Code Smell: Bloaters (Long Method)

– Where: selectPrintTask method in PrinterManager.java

– Whats wrong: selectPrintTask() method contains too many lines of code.

– Why is it wrong: Long method is difficult to understand and maintain. Furthermore, it provides an ideal hiding place for unwanted duplicate code.

– Possible solution: Extraction. This includes splitting up part of a selectPrintTask() method into separate method.

2. Refactoring

2.1 Applying Software Principles

1) Single Responsibility Principle

The Single Responsibility Principle, one of the SOLID principles of object-oriented design, states that a class should have one and only one reason to change, meaning it should have only one job or responsibility. This principle helps in making the system easier to understand and maintain.

Problem Statement:

The Main class that handles multiple tasks such as displaying menus, taking user input, reading data from a file, and updating printer status. This class violates Single Responsibility Principle because there is more than one reason for the change.

Design Solution:

Applying Single Responsibility Principle to Main.java class

Following the single responsibility principle, the Main class has been refactored to handle only one responsibility, which is to control (start/exit) the application.

**package** nl.saxion.app;  
**import** nl.saxion.app.facade.PrinterFacade;

**public** **class** Main {  
 **public** **static** **void** main(String[] args) {  
 **new** Main().run(args);  
 }

**public** **void** run(String[] args) {  
 PrinterFacade facade = **new** PrinterFacade();  
 **try** {  
 facade.readFromFiles(args);  
 facade.showMenu();  
 facade.exit();  
 } **catch** (Exception e) {  
 System.***out***.println("Error running application: " + e.getMessage());  
 }

}

}

2.2 Facade Pattern

The Facade Design Pattern in Java is a structural design pattern that provides a simplified interface to a complex subsystem or a set of interfaces in a system. This pattern introduces a facade class that acts as a single entry point, hiding the complexities of the subsystems from the client and facilitating easier access to the functionality of the system. The primary goal of the Facade pattern is to achieve a reduction in complexity for the client and to promote decoupling between the system and the client.

Problem Statement:

This code works, but it's cumbersome to manage individual components and their interactions.

Design Solution:

Introduce PrinterFacade class and Extract Main.java class code into it.

The facade class PrinterFacade.java is introduced to handle the complex processing of the 3dPrintScheduler application.

The complex functionalities handled by Printerfacade include:

* Display User Menu
* Read User Input
* Read Printer Data from Files
* Update Printer Status

3. Design Patterns

3.1 Singleton Pattern

The Singleton Design Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is used when exactly one object is needed to coordinate actions across the system.

Problem Statement:

The PrinterManager class holds application data loaded from a file. We want only one instance of this class in the entire application to ensure consistent data access.

Design Solution:

3.2 Factory Pattern

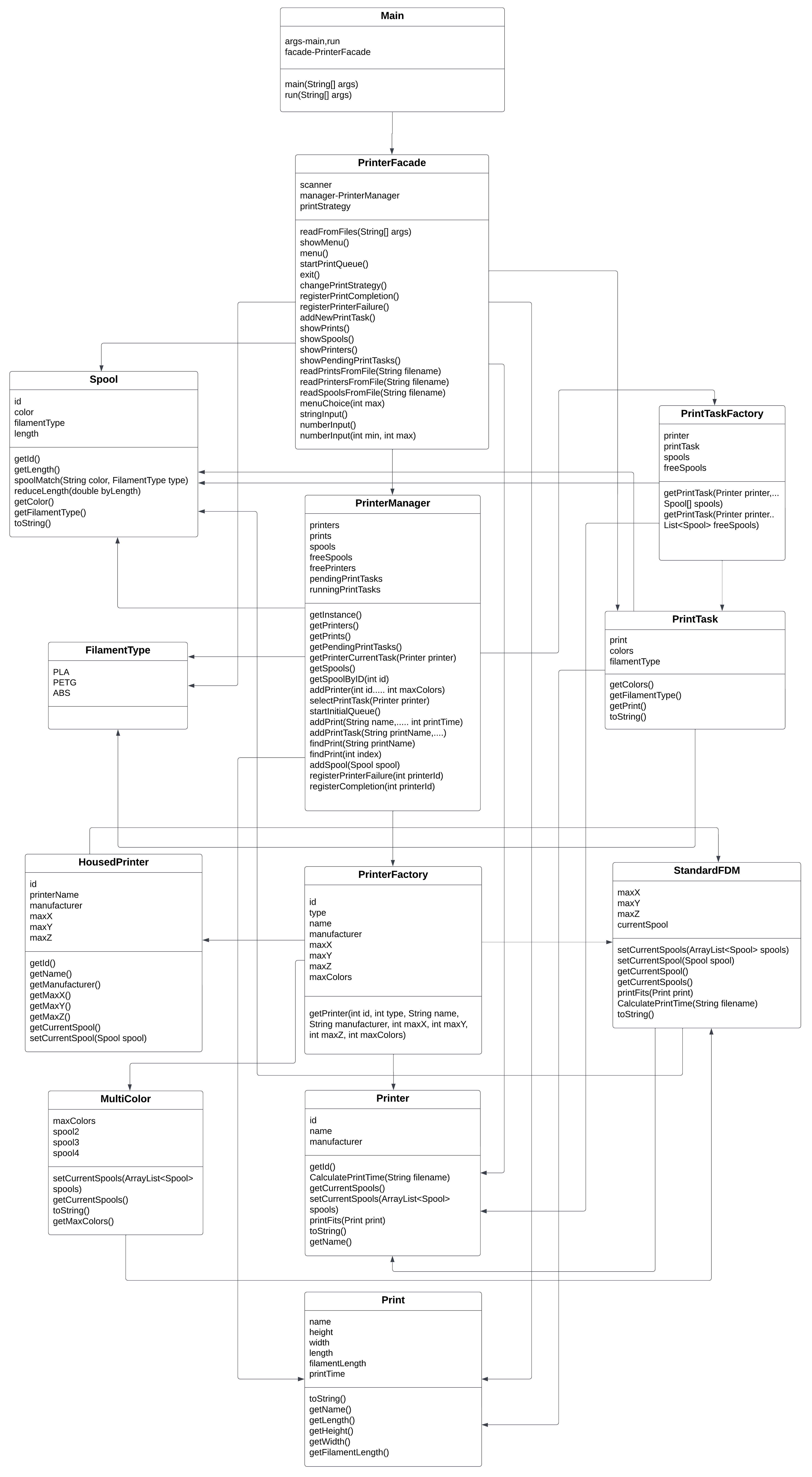
The Factory Design Pattern is one of the most commonly used design patterns in Java. It falls under the category of creational patterns, focusing on the process of object creation. Essentially, it delegates the instantiation of objects to subclasses, making the system more flexible and integrated.

Problem Statement:

Design Solution:

PrinterFactory.java added to ...

PrintTaskFactory.java added to ...



4. Expansion of the System

4.1 Expansion-1:

An additional option in the menu that displays the dashboard.

Following data will be displayed:

- Number of times a spool has been changed.  
- Total number of prints fulfilled.

5. Testing

6. Conclusion