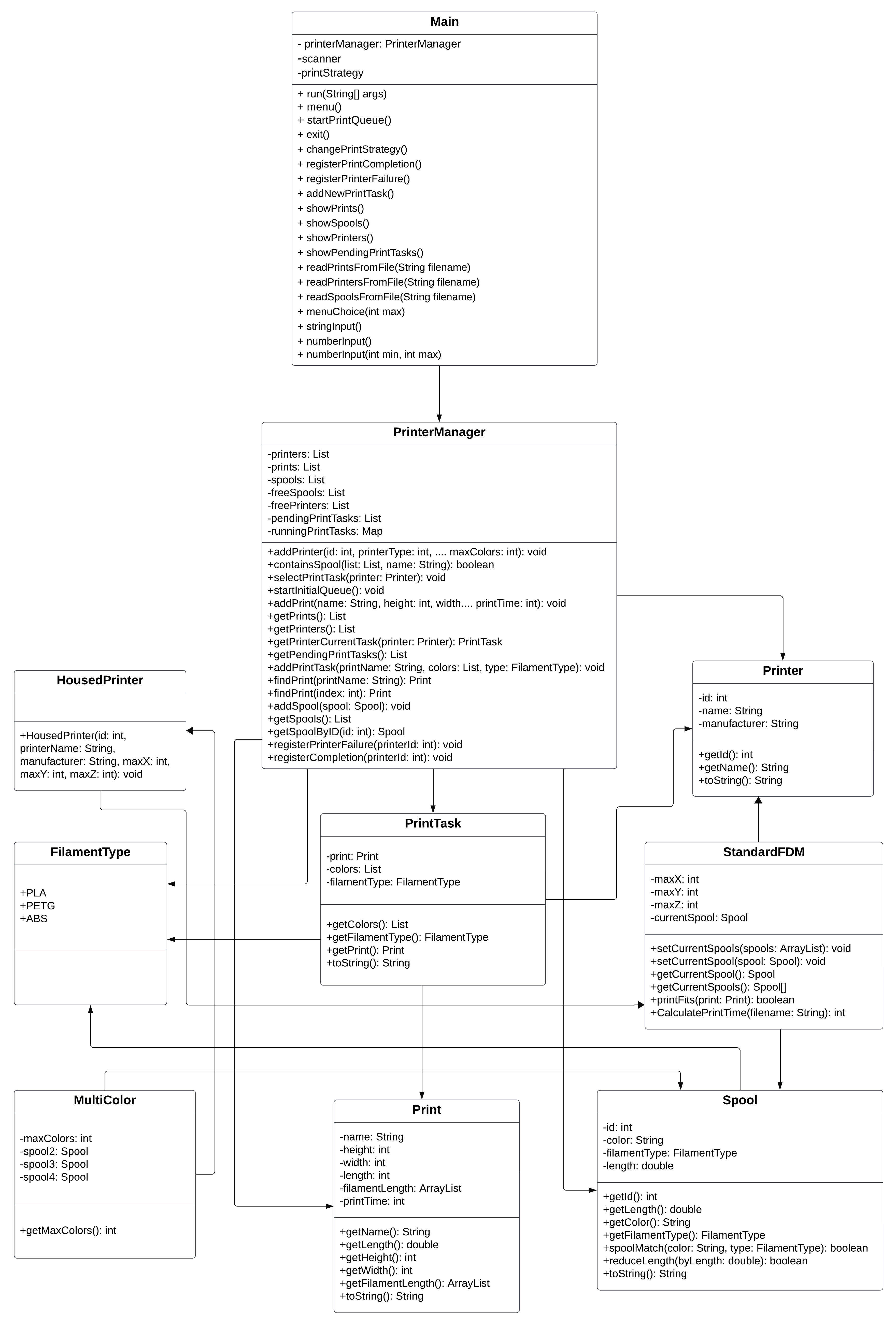
Technical Report

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:

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

Implementation:

Below is code snippet after applying refactoring the Main.java class

**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.

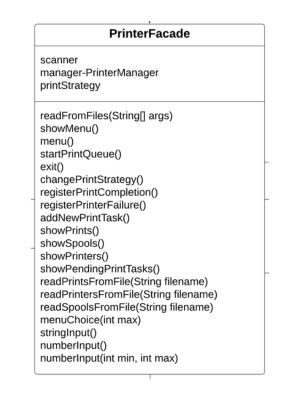
Design Solution:

Introduce a facade class and extract Main.java class code into it.

Implementation:

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:

Refactor PrinterManager.java to follow the singleton pattern.

Implementation:

Here, the PrinterManager class is refactored to follow the singleton pattern. The constructor is made private to prevent external instantiation. A static getInstance() method provides access point to get the single instance of the class. This ensures that only one PrinterManager object is created throughout the application.

// Singleton

**public** **class** PrinterManager {

// variables...

**private** **static** PrinterManager *instance*;

**private** PrinterManager() {  
 // Private constructor to prevent instantiation  
 }

**public** **static** PrinterManager getInstance() {  
 **if** (*instance* == **null**) {  
 *instance* = **new** PrinterManager();  
 }  
 **return** *instance*;  
 }

// methods...

}

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:

The PrinterManager.selectPrintTask method is tightly tied to specific printer types. This class needs to be modified to add the new printer type. As the number of printer types increases, the code becomes less readable.

Design Solution:

To solve this problem, we can use the Factory Design Pattern. We'll create a PrinterFactory class that defines a method for creating objects. The specific type of object to be created will be determined at runtime based on the input provided to the factory. This way, our system can decide which Printer type to instantiate without knowing the details of how each printer is created.

Implementation:

The PrinterFactory class responsible for creating Printer objects based on the type provided to the getPrinter() method.

Here's the implementation:

// Factory  
**public** **class** PrinterFactory {

**public** **static** Printer getPrinter(**int** id, **int** type, ……….) {

**if** (type == 1) {  
 **return** **new** StandardFDM(id, name, manufacturer, maxX, maxY, maxZ);  
 } **else** **if** (type == 2) {  
 **return** **new** HousedPrinter(id, name, manufacturer, maxX, maxY, maxZ);  
 } **else** **if** (type == 3) {  
 **return** **new** MultiColor(id, name, manufacturer, maxX, maxY, maxZ, maxColors);  
 }

**return** **null**;  
 }

}

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.

Design Solution:

The Observer design pattern is a behavioral design pattern that allows loose coupling between weather data and UI elements. It's very useful for implementing distributed event handling systems, in scenarios where an object needs to notify other objects about state changes.

Implementation:

Here, whenever the spool change count or the print completion count is updated, the DashboardManager class notifies the DashboardObserver, exposing the functionality of the Observer pattern.

5. Testing



This screenshot displays application output on the eclipse IDE console.

It shows:

* Application Main Menu
* Output of the command ‘*2) Register Printer Completion*’
* Output of the command ‘*10) Show Dashboard*’

6. Conclusion

The final project structure after refactoring the code by applying various design patterns and implementing system extensions is shown in the following class diagram.

