Technical report template

Part 1: Software modelling

# 1. Analysis

## 1.1 Class diagram

* *Create a class diagram from the 3dprintscheduler-main*
* *Note all the menu option actions the user can do in the 3Dprintschedular, which model classes are used for each action?*

### Class Diagram

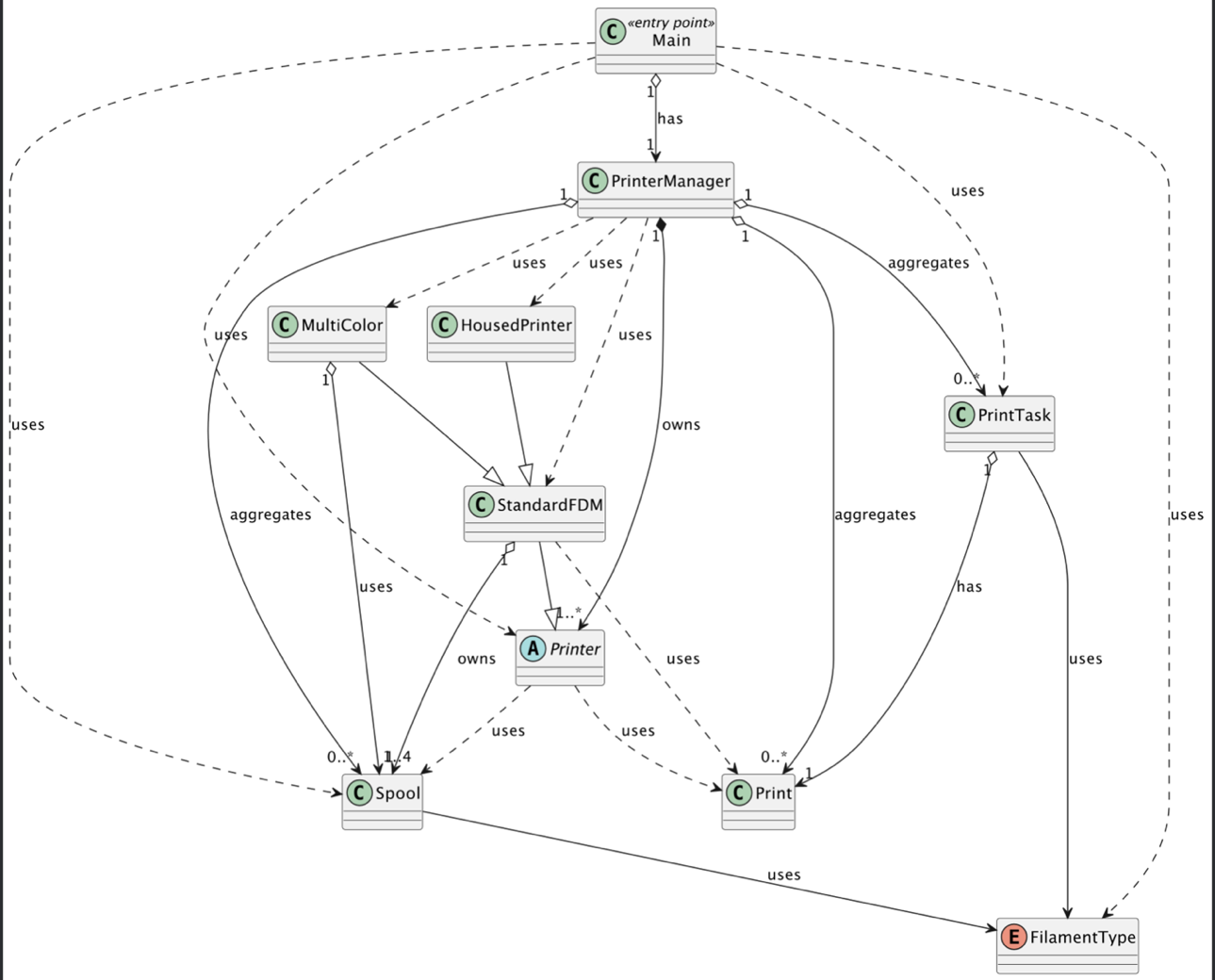
**

Figure 1: Initial Class diagram

The biggest problem of the system is classes being too dependent on each other’s implementation details. A change in one class would often require changes in another, making the system not scalable and difficult to maintain.

As illustrated in the diagram, examples of this issue are the Main and PrinterManager classes. The Main class is tightly coupled to PrinterManager, Spool, Printer, PrintTask, and FilamentType. If any of these classes (e.g., PrinterManager) changes its behavior or dependencies, the Main class might also require modifications.

This tight coupling means adding a new type of printer would require changes to PrinterManager. Moreover, PrinterManager aggregates too many responsibilities, such as managing printers, spools, and print tasks, which makes it a “God” class.

To sum up, covered issues make system hard to scale and test. For instance, testing Main or PrinterManager independently is challenging since they rely on multiple concrete implementations.

### Menu Options

|  |  |  |
| --- | --- | --- |
| Menu Option | Explanation | Dependent Classes |
| (1) Add new Print Task | Allows the user to add a new print task by selecting a print, filament type, and colors | Print, FilamentType, Spool, PrinterManager |
| (2) Register printer completion | Prompts user to select specific printer id that has completed print task | Printer, PrintTask, PrinterManager |
| (3) Register printer failure | Prompts user to select specific printer id that has failed print task | Printer, PrintTask, PrinterManager |
| (4) Change printing style | Prompts user to select new print strategy, but actually does not do anything. Was left purposefully to add this feature in future | None |
| (5) Start print queue | Starts the initial print queue managed by the PrinterManager | PrinterManager |
| (6) Show prints | Displays all available prints | Print, PrinterManager | Print, PrinterManager |
| (7) Show printers | Displays all available printers and their current tasks | Printer, PrintTask, PrinterManager |
| (8) Show spools | Displays all available spools | Spool, PrinterManager |
| (9) Show pending tasks | Gets all print tasks from printer manager and displays them | PrintTask, PrinterManager |
| (0) Exit | Stops the infinite loop for reading user input and closes application | None |

## 1.2 Code smells

*Use the following format, see if you can find at least 3-5 code smells, depending on how impactful they are*

* *Where:* In Main class, the run() method
* *What’s wrong:* The method has a very long sequence of if-else statements to handle user menu option choice.
* *Why is it wrong:* It is hard to read through the code and makes it harder to maintain options.
* *Possible solution:* Use a switch case instead.
* *Name of code smell:* Long method
* *Where:* In Main class, the addNewPrintTask() method
* *What’s wrong:* This method consists of a very long sequence of conditions and nested loops
* *Why is it wrong:* Similar to the previous example, it is hard to read through the method and understand the logics.
* *Possible solution:* Separate the concerns, by splitting parts of the method into separate smaller functions.
* *Name of code smell:* Long method
* *Where:* InMain class, methods readPrintsFromFile(), readPrintersFromFile(), and readSpoolsFromFile()
* *What’s wrong:* Each method contains similar code for reading JSON files and parsing their contents.
* *Why is it wrong:* Duplicate code is harder to maintain, as well as harder to read through the Main class
* *Possible solution:* Extract method into separate method outside the Main class. I see the possibility of using a generics and functional interfaces for different object parsing. A great example of this is implementation of LambdaReader from SDP module, where we have used Function interface to turn the list of columns into desired Class type.
* *Name of code smell:* Duplicate Code
* *Where:* The Main class itself
* *What’s wrong:* A class have to many responsibilities, including file reading, user input handling, handling menu options etc.
* *Why is it wrong:* Hard to read, understand and maintain
* *Possible solution:* Separate functionality between different classes, trying to adhere to Single Responsibility Principle
* *Name of code smell:* Large Class
* *Where:* PrinterManager class, the addPrint() method
* *What’s wrong:* The method accepts 7 different parameters.
* *Why is it wrong:* Similar to previous “code smells”, this large amount of parameters makes it harder to read and, of course, maintain.
* *Possible solution:* Accept one parameter of Print class
* *Name of code smell:* Long Parameter List
* *Where:* PrinterManager class, the getSpoolByID() method
* *What’s wrong:* The method is not used in the program
* *Why is it wrong:* It is unnecessary to keep unused methods
* *Possible solution:* Remove method
* *Name of code smell:* Dead code
* *Where:* PrinterManager class, the registerCompletion() and registerPrintFailure() methods
* *What’s wrong:* Both methods contain similar code for finding a printer by ID and handling the print task.
* *Why is it wrong:* Duplicate code increases maintenance effort and the risk of inconsistencies
* *Possible solution:* Extract duplicate code into a separate method
* *Name of code smell:* Duplicate Code

## 1.3 Structural problems in the code

*Write down bigger structural problems you found in the code (at least 2 or more). These problems often involve multiple classes. Write down where the problem lies and why this is a problem. Optional: Also, show diagrams to make it clearer.  
  
Hint: are the printer classes logical (and needed)?*

#### Printers Inheritance

The current printer inheritance structure consists of an abstract class, Printer, which serves as the base for all printer types. StandardFDM inherits from Printer and represents a standard FDM printer. The hierarchy is then divided into specific types: HousedPrinter, which extends StandardFDM to represent housed printers, and MultiColor, which also extends StandardFDM to, as the name implies, handle multicolor printing.

One significant problem is the depth of the inheritance hierarchy. Changes to Printer or StandardFDM could potentially break functionality in HousedPrinter or MultiColor.

This close relationship in the hierarchy not only increases the complexity of the maintenance, but also increases the chances of unexpected side effects.



Figure 2: Printer inheritance

#### HousedPrinter Redundance

In my opinion, having a HousedPrinter class is redundant. As you can see on the diagram above, it doesn’t actually extend the StandardFDM by any means (e.g. no specific class field or methods), making this inheritance relationship irrational.

It is clear for me that developer initially created this class to have a distinct printer type and might have had some additional functionality planned for it. However, as it does not contain any meaningful functionality, this could have been avoided by having a class field, such as “isHoused” in StandardFDM.

#### Package Structure

Currently the system has one root package, containing Main and PrinterManager classes which and a sub-package “Models”, which contain all the application used classes. This sub-package, however, does not follow any rules in grouping classes. All of the classes coexist in the same place.

The inconsistency in package structure can cause confusion about where to find or place new classes. It can also make it harder to understand the organization of the codebase.



Figure 3: Package structure

# 2. Design

*This chapter is about your design, so everything in here should be done BEFORE you start coding!*

## 2.1 Code improvements and SOLID principles

*Design better code by applying the SOLID principles and other improvements. Show diagrams that show how the new structure/code would look and explain your solution.*

#### Applying Single Responsibility principle

Right now, all the core application methods are mainly distributed between Main and PrinterManager class, making them God classes. As you can see on the diagram (Figure 4), they do “everything”, there is no separation of concerns.

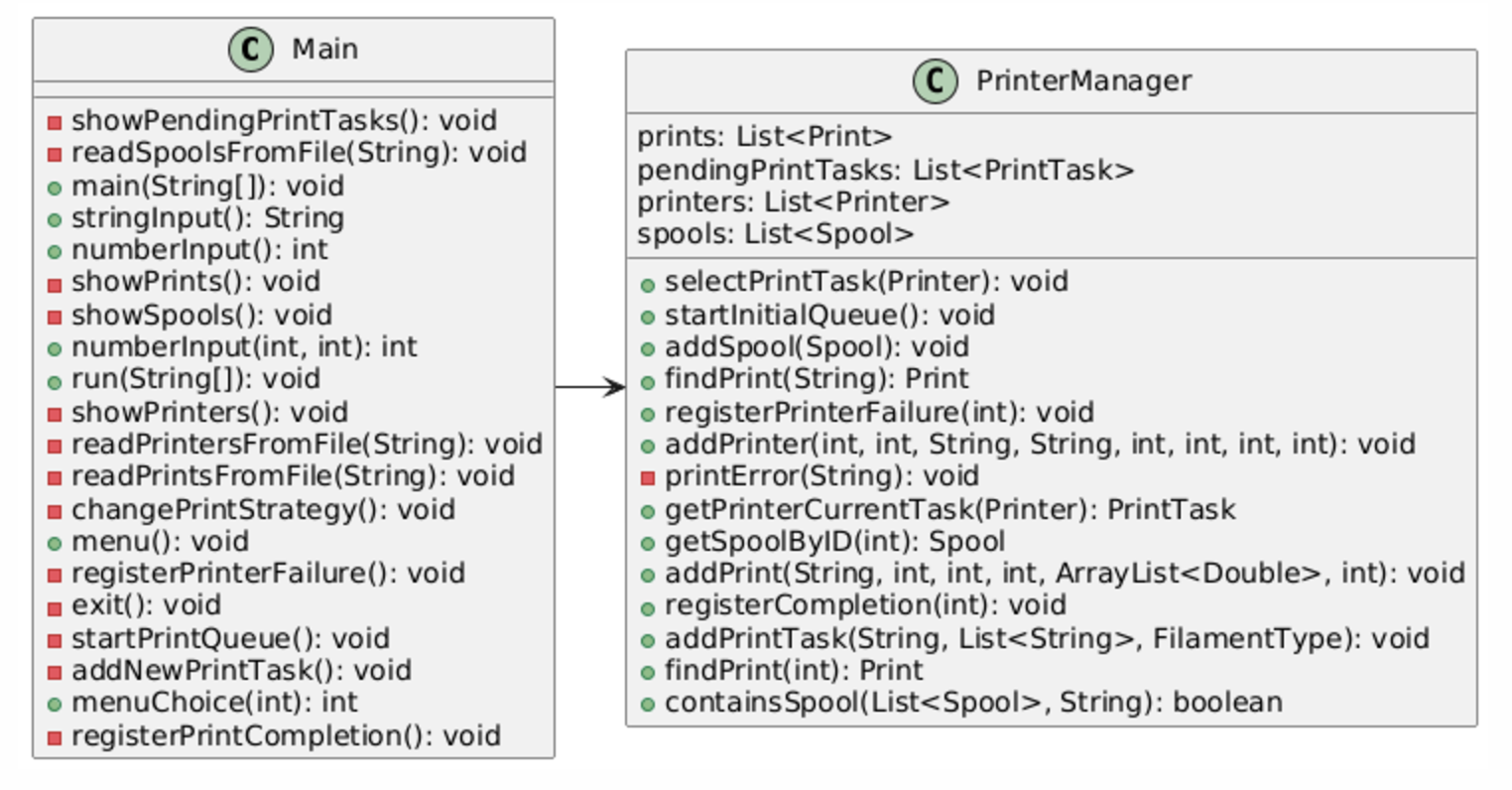


Figure 4: Main and PrinterManager classes having too many concerns

As Single Responsibility rule states, a class can have only one responsibility. This not only improves the organization of the system and avoids class coupling, but also allows us to test Application better. We can obviously apply this principle to both Main and PrinterManager.

When it comes to Main, it should only start the application, by initializing all the components of the system and not manage menu, input and file reading. We will introduce:

* UserInputHanlder, which will handle console operations for string and number input
* MenuManager, which will display the menu and handle user’s choice
* FileReader, which will manage file operations

As far as PrinterManager is concerned, we can separate concerns into small, specialized classes:

* PrinterHandler – handles printers
* PrintTaskHanlder – handles print tasks
* SpoolManager – manages spools

In our opinion, this change will allow to handle concerns better. On a following diagram (Figure 5), we show what are the system components after applying the principle. There are, obviously, more changes coming, as we continue to apply SOLID principles. Thus, not all methods from the previous diagram are shown and no association is drawn. This diagram exists purely to depict that God classes of the system are separated into smaller classes, each responsible for one functionality.

A screenshot of a computer program

Description automatically generated

Figure 5: Classes after applying SRP principle

#### Applying Open-Closed Principle

Right now, Application depends only on console input and JSON File reading. When it comes to user input, we understand that current setup provides fast usage to the owner, however, what if in future, as his company grows, there would be a need in GUI? Or, from testing perspective, it is not really possible to implement Unit-testing for input right now.

Similarly, now program relies on JSON, however, what if there should be CSV file support? Or perhaps, owner will save the important data in the database?

Current setup is not open for extension, as adding new type of input or data source, will require code modification. This can be fixed creating specific interface per problem. Having an interface for both user input and data reading will allow to have multiple sources of truth. Whether it is a mock input, GUI input or console input, the application logic won’t need to be modified.

This will allow to extend and test application more easily.

On the diagram (Figure 6), we have displayed possible classes, that can easily the extend system with this setup. JSONFileReader class, depicted on the diagram, would handle the current logic for reading the data out of Json files.

A screenshot of a computer

Description automatically generated

Figure 6 Applying Open-closed for input and data reading

#### Applying Interface Segregation

The previous example of user input and data reading can be further improved, if we would

apply Interface Segregation principle for better flexibility. This is relevant for IDataProvider

interface from the Figure 6 diagram. This is not really flexible and can be improved.

As we would implement large interface, we would have to implement all of the existing

interface methods, which is not always the case.

On a Figure 7, we show, what is the advantage of splitting the interface. With this setup, we

can have different implementation of Data providers. One class can implement multiple data

provider interfaces, allowing for flexible usage and different combinations.

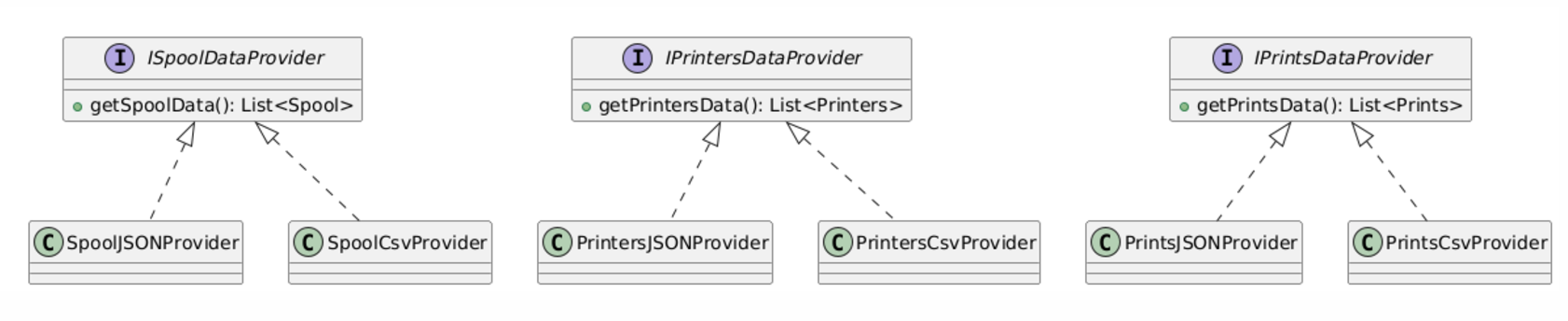


Figure 7 Example usage

On the following figure, we display how we might handle getting data in the Application now. (This might change after applying Design Patterns)

A screenshot of a computer

Description automatically generated

Figure 8 A possible solution relevant to current setup

## 2.2 Design patterns

*Discuss which design patterns you apply and why these would improve your code. Make sure to include diagrams to make it clearer. Hint: see if you can apply at least 3-5 design patterns*

*Ask yourself these questions for each design pattern:*

* *What type of problems do you have in the code?*
* *Is there a design pattern that solve this problem?*
* *How does this design pattern solve this problem?*
* *Show/discuss how you going to apply this pattern*
* *Convince the reader that you made the code better*

#### 2.2.1. Adapter Pattern

***Issue:*** *The original code tightly coupled the JSON parsing logic with the application logic, violating the Single Responsibility Principle and making it difficult to:*

* *Extend support to new data formats (e.g., XML, CSV).*
* *Maintain or modify the parsing logic without affecting the core logic.*
* *Test the application independently of the data format.*

***Fix:*** *Applied the Adapter Pattern to decouple data sources from the application logic and standardize data handling:*

* *All data sources (JSON, XML, CSV, etc.) are now converted into a common HashMap format. This is done via* “ISourceAdapter” *implementations (e.g.,* “JsonAdapter”*).*
* *Each model (Print, Spool, Printer) now includes a* “fromMap” *method, which encapsulates the logic for creating an instance of the model from a HashMap. This simplifies object creation and centralizes it within the models.*
* *The core application logic interacts only with the* “ISourceAdapter” *interface, which ensures data is provided in a consistent HashMap format, regardless of the original source.*

*Diagram 9 illustrates how adapters transform various data sources into a unified format that the application can easily consume.*

*A diagram of a computer system

Description automatically generated*

*Figure 9. Adapter Pattern*

***Justification:*** *This approach excels due to its modularity. By isolating responsibilities, the system achieves high flexibility and maintainability. Adapters focus solely on transforming data formats, while models handle object creation via a standardized process. The core logic remains entirely abstracted from the specifics of data parsing or source format.*

#### 2.2.2 Factory Pattern

***Issue:*** *The original code tightly couples the logic for reading and creating “Printer” objects with the “PrinterManager” class. This causes several issues:*

* *The* “addPrinter” *method in* “PrinterManager” *is overloaded with the responsibility of creating different types of Printer objects.*
* *Adding a new printer type (e.g.,* “ResinPrinter”*) requires modifying the* “addPrinter” *method, which violates the Open/Closed Principle (OCP).*
* *The creation logic for printers is embedded into* “PrinterManager”*, making it hard to reuse in other parts of the application.*

***Fix:*** *Applied the Factory Pattern to decouple printer creation logic from the “PrinterManager” and integrate it with “Mapper”:*

* *Introduced a* “PrinterFactory” *class to encapsulate the logic for creating* “Printer” *objects.*
* *The* “PrinterFactory” *class determines the correct type of* “Printer” *to create based on the “type” field, ensuring consistent and reusable logic.*
* *Removed direct printer creation logic from* “PrinterManager” *and delegated it to the* “PrinterFactory”*. The* “Mapper” *now directly utilizes the factory to create “Printer” objects from data.*

***Justification:*** *The Factory Pattern centralizes printer creation logic in*“PrinterFactory”*, making the system modular, maintainable, and extensible. Adding new printer types requires changes only in the factory, adhering to the Open/Closed Principle, while*“Mapper”*and*“PrinterManager”*remain focused on their respective responsibilities. This separation ensures consistency, reduces duplication, and simplifies future modifications, making the code cleaner and easier to manage, as illustrated in Figure 10*

*A diagram of a computer program

Description automatically generated*

*Figure 10. Factory-based creation of Print objects.*

#### 2.2.4 Singleton Pattern

***Issue:*** *The current implementation of* “PrinterManager” *is responsible for storing and managing all data related to printers, spools, and tasks. Multiple instances of “PrinterManager” can be created, leading to inconsistent resource management and potential duplication of effort when interacting with printers and tasks. This undermines the concept of a centralized system for managing these resources.*

***Fix:***

* *Made the* “PrinterManager” *constructor private to prevent external instantiation.*
* *Introduced a static getInstance method to ensure only one instance of* “PrinterManager” *is created and shared across the application.*
* *Ensured that all operations, such as adding printers, managing spools, and assigning tasks, are performed on a single centralized instance.*
* *Replaced direct instantiations of* “PrinterManager” *in* “Facade” *with* “PrinterManager.getInstance()” *to guarantee consistent access to the shared instance.*

***Justification:*** *The Singleton Pattern directly addresses the problem by ensuring that only one instance of* PrinterManager *exists, centralizing resource management and preventing redundant operations. This approach avoids conflicting states across multiple instances and guarantees consistent behavior. By ensuring all parts of the application interact with the same instance, the design reduces complexity and improves maintainability. This diagram*

*(Figure 11) illustrates the Singleton Pattern applied to* PrinterManager*, ensuring a single, shared instance accessed globally through Facade and preventing multiple conflicting instances.*

*A diagram of a computer program

Description automatically generated*

*Figure 11. Singleton Pattern implementation for centralized* PrinterManager *instance*

#### 2.2.5 Facade Pattern

***Issue:*** *The original implementation of* “Main” *directly handles logic for user input, file management, and interactions with* “PrinterManager”*. This creates several issues:*

* “Main” *integrates user input handling, file operations, and printer management, violating the Single Responsibility Principle.*
* *“Main” is overly large, mixing different concerns, making it difficult to maintain or extend.*
* *Tasks like menu display and command processing are implemented redundantly, complicating code reuse.*

***Fix:***

* *Created* “Facade” *class to encapsulate all interactions with* “PrinterManager” *and related components (e.g., file reading, print queue management).*
* *Moved all high-level operations, such as* “addPrintTask” *or* “startPrintQueue”*, into the Facade.*
* *Added* “UserInput” *class to handle all user input operations (e.g., reading integers, strings, and menu choices).*
* *Simplified the input-related logic in* “Main” *by delegating it to* “UserInput”*.*

***Justification:*** *As illustrated in Figure 12, the application now uses the Facade Pattern to provide a unified interface for managing* PrinterManager *and related components. This reduces coupling, improves code clarity, and aligns with the Single Responsibility Principle. The addition of* “UserInput” *ensures cleaner, reusable input handling. This design simplifies the application, making it easier to extend and maintain by isolating user interaction, core logic, and resource management.*

*A diagram of a computer program

Description automatically generated*

Figure 12 Facade integration for simplified management of PrinterManager and related components

#### 2.2.6 Strategy pattern for task selection

**Issue:** The original method for selecting the print task is rather enormous, spanning over 112 lines of code. The method is brute forcing the logic and lacking code structure, readability and is really prone to errors, when it comes to modification or extension.

**Fix:** We can simplify the logic of the method and separate concerns more by applying strategy pattern in this method.

First, we create an interface PrintTaskSelector, which has selectTask method. After that, we create a class for each specific selection strategy, that implement this interface.

A screenshot of a computer program

Description automatically generated

Figure 13 Strategy Pattern

We can then add a method to an Abstract printer class getPrintTaskSelector(), which returns a PrintTaskSelector instance of respective select strategy implementation. On the following figure, there is an example usage for StandardFDM. It will of course return a new instance of StandardFDMPrintTaskSelector from the previous [Figure](#StrategyPattern).



Figure 14 Example usage

By applying the Strategy Pattern, the selectPrintTask() method is simplified, and the logic for selecting print tasks is encapsulated within the appropriate strategy classes.

The implementation would probably look similar to this code snippet:

public void selectPrintTask(Printer printer) {  
 PrintTaskSelector selector = printer.getPrintTaskSelector();  
 PrintTask chosenTask = selector.selectTask(printer, pendingPrintTasks, freeSpools);  
  
 if (chosenTask != null) {  
 pendingPrintTasks.remove(chosenTask);  
 runningPrintTasks.put(printer, chosenTask);  
 freePrinters.remove(printer);  
 System.*out*.println("- Started task: " + chosenTask + " on printer " + printer.getName());  
 }  
}

Of course, we would have to modify the PrinterManager class to store the instance of PrintTaskSelector as well.

#### 2.2.7 Strategy selection using Strategy Pattern

**Issue:** The current method “changePrintStrategy” acts as a placeholder, and author left a comment expressing an intention of having such functionality in the future. No actual logic is implemented for each strategy.

**Fix:** Although it is a future consideration, we have decided to design the possible look of this feature in our system. This is a perfect use case for strategy pattern.

A diagram of a strategy

Description automatically generated

Figure 15 Print strategies

The approach to applying the Strategy pattern is similar. We create an Interface IPrintStrategy and two classes, which implement it accordingly.

In the PrinterManager class, we can now change the strategies on the runtime by having a setter for a IPrintStrategy instance

## 2.3 Global overview

*After applying all your solutions how does your code look? Show a global class diagram. Explain/discuss areas that might require a bit more explanation.*

*hint: What information should you show and what information is too detailed? What are you trying to show here?*

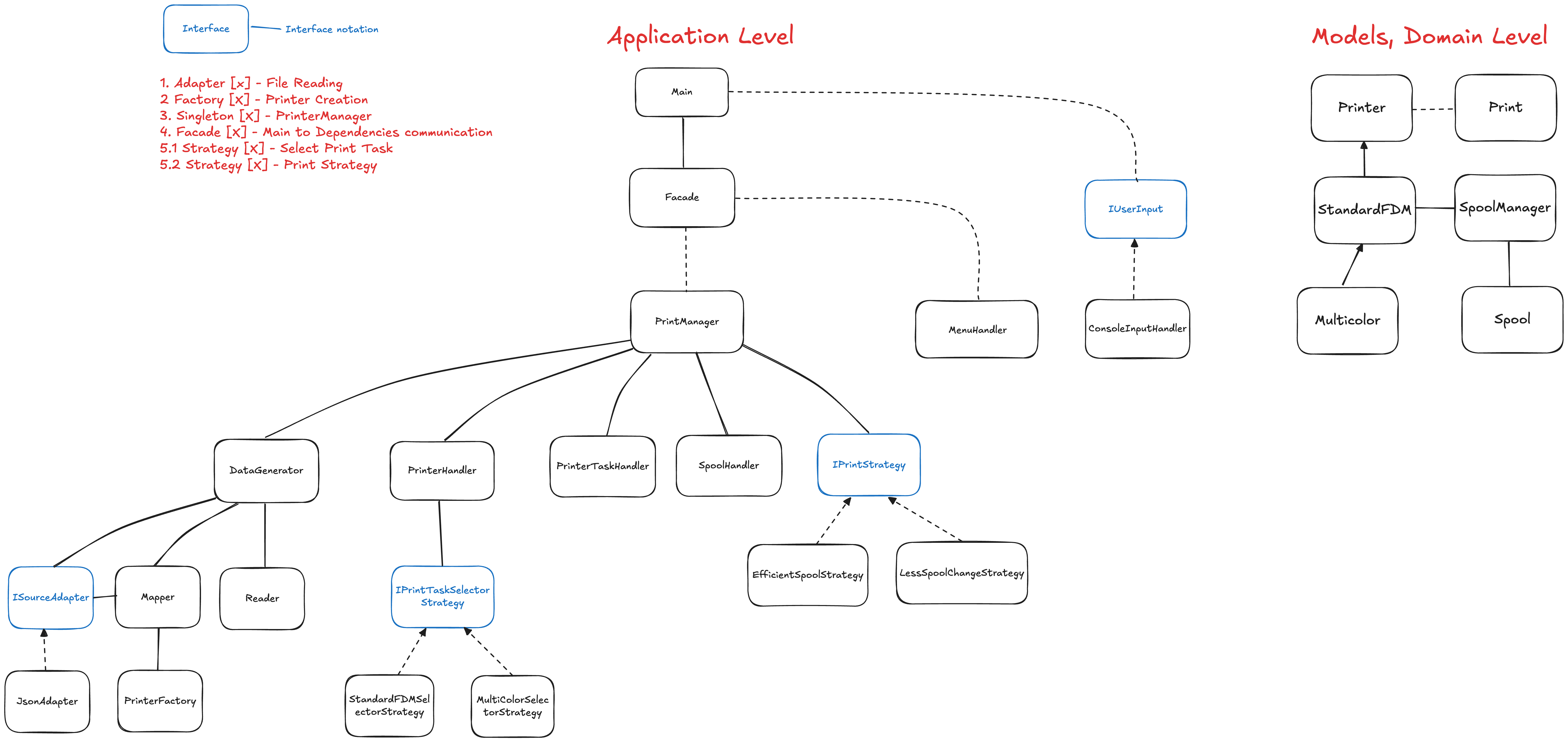


Figure 16 A sketch of our design. Application level related classes and models are split accordingly.

On the given diagram, we present a sketch of a system after applying the design patterns and SOLID Principles. A box with blue stroke represents an interface. We have used UML

standards for drawing relationships. However, for the sake of making diagram not as

cluttered, considering the amount of new introduced classes and interfaces, we have

not modeled the relationship between Application classes and Model classes

One thing that needs more clarification is the Model classes that we presented on the right of Figure 9. We decided to change the current inheritance chain of printers a little bit. As previously mentioned, HousedPrinter is redundant and will be replaced with isHoused Boolean value.

We have introduced a new class SpoolManager though, that will handle the spool management more efficiently. Current logic related to managing spools is odd and not flexible. What if we want to add another spool to multicolor?

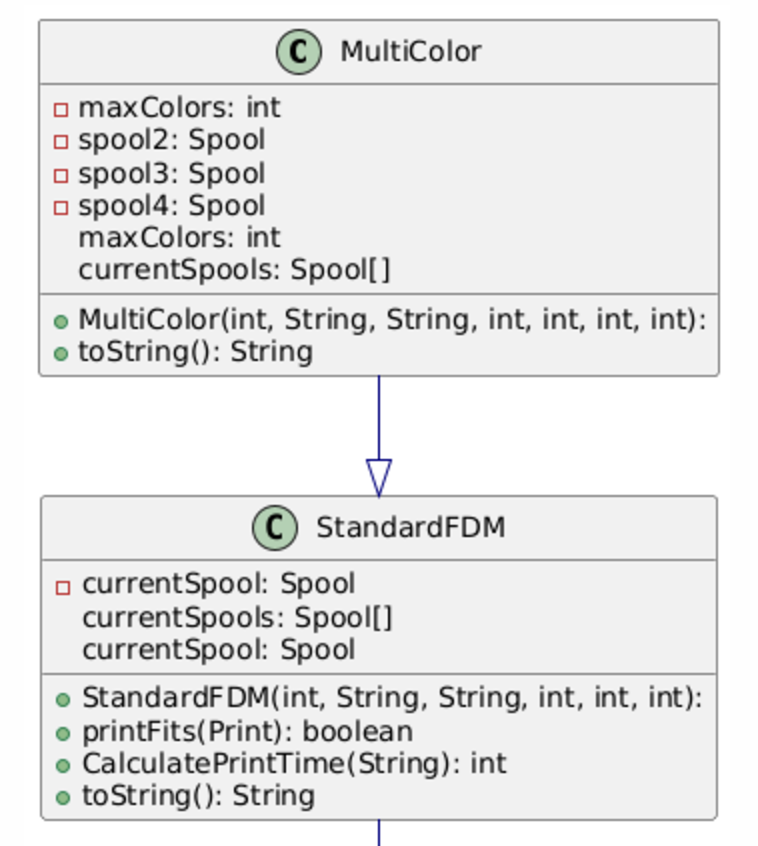


Figure 17 Odd-ball solution to managing spools now

A SpoolManager will try to provide abstraction on managing spools in a more centralized way.

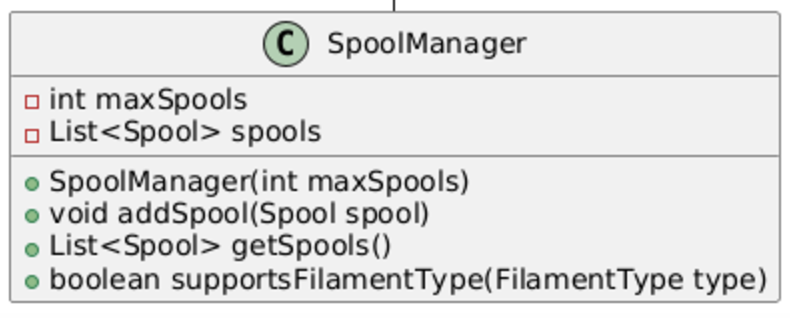


Figure 18 SpoolManager

Also, as presented on the diagram, we have used an Interface for user input adhering to Open-Closed principle. For now, we will use only ConsoleInputHanlder class, but with this setup we are able to “inject” other type of inputs in the future.

# 3. Implementation

## 3.1 Implement your design

*Implement your designs from Chapter 2. In this chapter write down the things you encountered. For example: did you have to change your design, did you make mistakes? Did you gain new insights?, etc.*

## 3.2 Extension of the system

*Implement the new extensions. Write down the problems you encountered and how you solved them. For example: What and why did you have to change your designs?*

## 3.3 Testing

*Include proof that your code is correct, you can do this with software testing. Add the results to this chapter.*

# 4. Conclusion

*Add a (UML) global class diagram of your final design. Make sure there is enough justification.*

*Hint: think about how much detail should you show*

Part 2: Software architecture

# 5. Software architecture

Goal: Create architectural design for the ***3D printer case part 2****!* Your goal is to convince the reader that your architectural design could work for this business case!

At least do the following:

* *Create a use case diagram*
* *Create a sketch of the system*
* *Create a deployment+component diagram*
* *Add extra information to elaborate on your choices*
* *Create an activity diagram for requesting a 3D print*
* *Add connection types to your component/deployment diagram*
* *Consider adding the architectural patterns discussed in week 6.2*
* *Discuss the architectural patterns*
  + ***Not in general, but how you applied them to your casus!***
* *Create at least (in total) 2 sequence diagrams for the most important actions*
* *Discuss alternatives you decided against and why!*
* *Convince the reader about your design!*
* *Consider potential future expansions of the system and the measures you will take to accommodate them*