Technical report template
Part 1: Software modelling

Table of contents:

1. Analysis

1.1 Class diagram

For this part we divided the whole diagram into smaller class diagrams, so that at the end the whole class diagram would be a product of the individual diagrams merged together. This made it easier for us to grasp what the code is doing, and explain it, and see the issues at smaller levels.

The diagram from Figure 1.0 shows all the relations of Printer. Printer is shown at the top and it is an abstract class. StandardFDM extends Printer, hence it StandardFDM is a child of Printer. HousedPrinter and MultiColor are children of the StandardFDM, indirectly children of Printer. Main uses Printer in its implementation. This shows how the classes are related and built on each other.

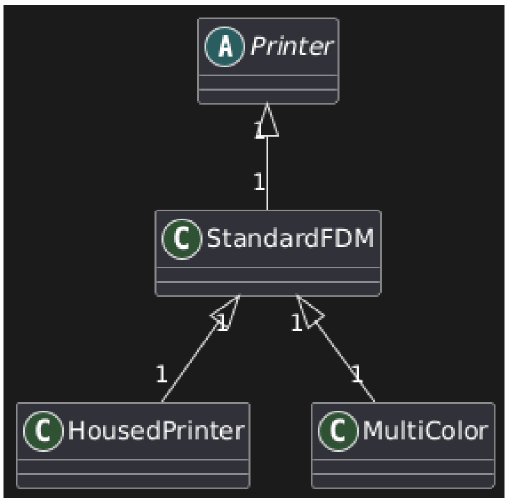


Figure 1: Printer class and its associations

In *Figure 2* (see below), all the relations from Spool are shown. Main, PrinterManager, MultiColor, Printer, and StandardFDM are the classes the Spool class associates with. Each of these has a specific relationship with Spool, with some being one-to-one (1) and others allowing multiple connections (*). This shows how Spool interacts with the other classes.

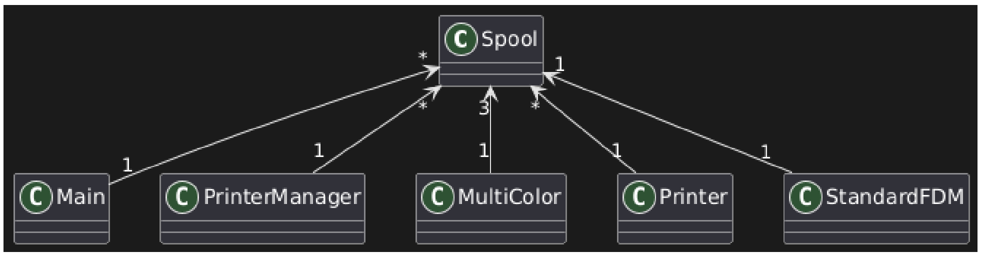


Figure 2: Spool class and its associations

In *Figure 3* the relations from Print are shown. In PrinterManager and Main there are many prints instances, they are in use. In Printer there is one print as well as in PrintTask

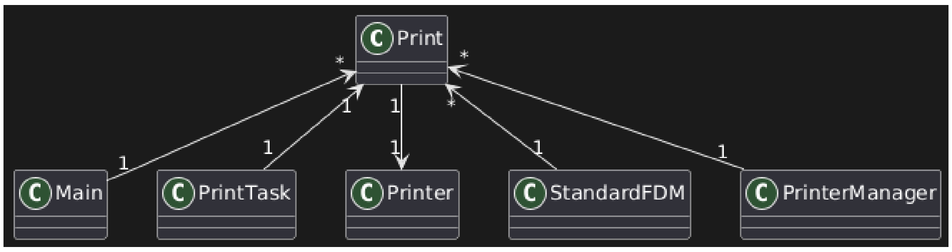


Figure 3: Print class and its associations

This diagram below - *Figure 4* shows the Main class connecting to several other components: PrinterManager, PrintTask, PrinterManager, Print, Printer, FilamentType, and Spool. Most relationships are one-to-one (1), meaning Main interacts with a single instance of each, except for Spool and Print, which are one-to-many (*), allowing multiple spools. This illustrates how Main coordinates tasks, printers, filament types, and spools in the system.

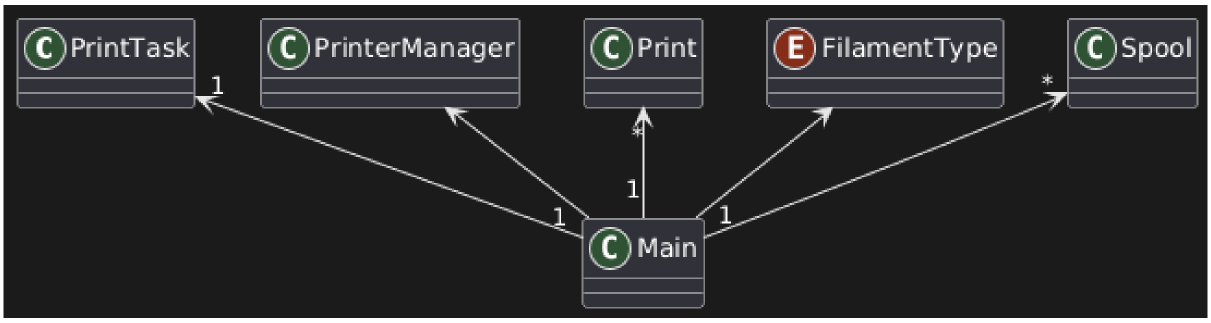


Figure 4: Main and its relations

The diagram in *Figure 5*, depicts the relationship between the PrinterManager and the classes. As it can be seen from the diagram, there are aggregation [1]relationships between most of the classes and the PrinterManager class. These relationships are very evident from the PrinterManager class entities, there are shown with the private list variables of Printers, Prints, PrintTasks, Spools.

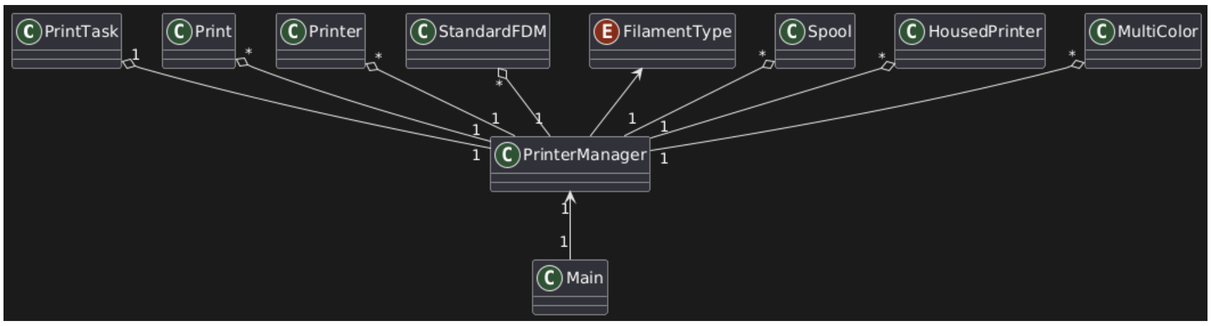


Figure 5: PrinterManager and its relations

Finally, the whole class diagram including all the classes is shown in *Figure 6*. It consists of multiple classes like PrinterManager, Print, Spool, FilamentType, the classes are meant to work togetger to support the system's functionality. The idea is that PrinterManager consists of

[1] Aggregation relationship – 'Indicate that instances of one model element are connected to instances of another model element' (IBM, 2021)

different printer types, such as StandardFDM, MultiColor, and HousedPrinter, which handle specific configurations and connect to resources like Spool and FilamentType.

The diagram also highlights the dependencies between these components, showing how printers rely on spools and filament type to function and how tasks are managed through PrintTask and Print. The Main class depends on six classes—PrintTask, PrinterManager, Print, Spool, Printer, and FilamentType to and manage the entire 3D printing process. Currently, the diagram looks tedious and messy, which is the result with the given code.

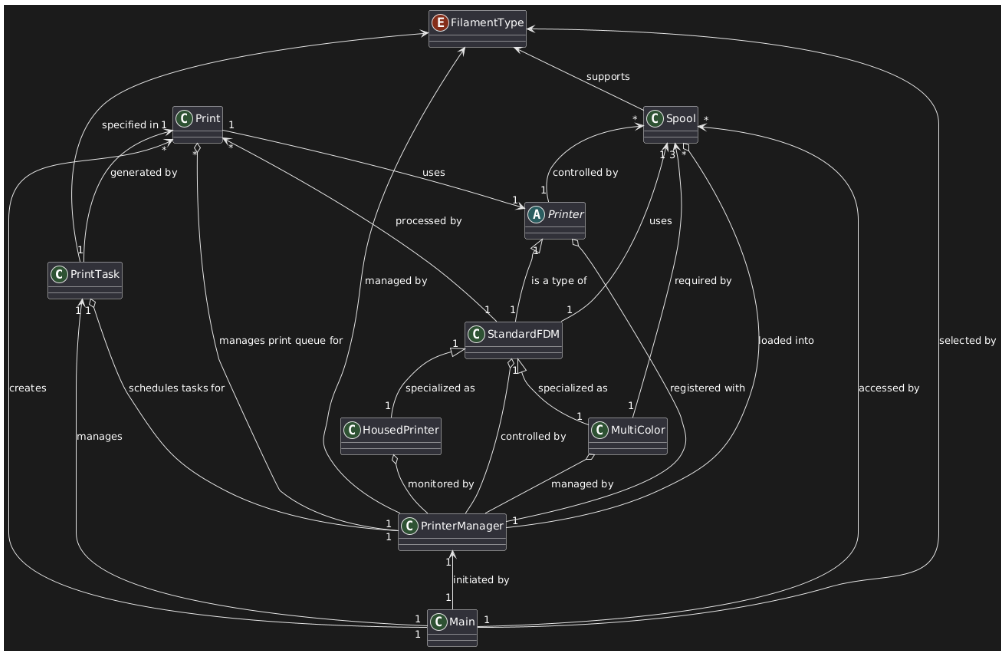


Figure 6: Main Diagram and all relations

Menu Option	**Model class used**
("- 1) Add new Print Task"	**FilamentType,Print,Spool,PrintTask**
("- 2) Register Printer Completion"	**Printer,PrintTask,Spool**
("- 3) Register Printer Failure"	**Printer,Spool,PrintTask**
("- 4) Change printing style"	NA
("- 5) Start Print Queue"	**Printer,PrintTask,Spool,*StandarFDM, HousedPrinter,MultiColor***
("- 6) Show prints"	**Print;**
("- 7) Show printers"	**Printer,PrintTask;**
("- 8) Show spools"	**Spool;**
("- 9) Show pending print tasks"	**PrintTask;**

Option 1 will start by getting a list of all the **prints (Print)** to get the Printer name then it will ask what **Filament type** it want to be used, then a list of **Spools (Spool)** for the chosen color and at the end the **PrintTask** will be called for a new task to be created.

Option 2 will start by getting a list of all the **prints (Print)** to get the running Prints and for each Print **PrintTask** will be called to print its running Tasks, ending with the method registerCompletion from PrinterManager called making **Spool** called to select the Print Task

Option 3 will start by getting a list of all the **prints (Print)** to get the running Prints and for each Print this task from **PrintTask** and if an error occurs the registerPrinterFailure method from PrinterManager is called making **Spool** called to select the Print Task.

Option 4 does not use any models' classes.

Option 5 starts by starting the Prints queues by calling the startInitialQueue method from PrinterManager and in here all **prints (Print)** call selectPrintTask that will get the **Spools** from that print and for selecting the coorect Task will loop for all the **PrintTaks** and using the **StandarFDM**, **HousedPrinter** and **MultiColor** to achieve it.

Option 6 will start by getting the list of all **Prints** and displaying them.

Option 7 will start by getting the list of all **Printers** and displaying them by using the method in PrinterManagger called getPrinterCurrentTask that will get the **PrintTask**.

Option 8 will start by getting the list of all **Spools** and displaying them

Option 9 will show a list of all the **PrintTask** that are occurring.

1.2 Code smells

1.2.1 Code smells

- *Where:* The selectPrintTask method in the PrinterManager class.
- *What is wrong:* The selectPrintTask method is very long and overly complex, making it difficult to follow and understand.
- *Why is it wrong:* Methods that are too long and complex are harder to read 150 lines, understand, and maintain. This increases the chance of errors and makes debugging more challenging.
- *Possible solution:* Break the selectPrintTask method into smaller, simpler methods that focus on specific parts of the task selection process.
- *Name of code smell:* Bloaters – Long Method

1.2.2 Code smells

- *Where:* The Main class
- *What is wrong*: The Main class is too big. Specifically, the run method does not follow the DRY principle and calls three long methods: readPrintersFromFile, readPrintsFromFile, and readSpoolsFromFile. Additionally, these methods don't belong in the Main class and repeat the same checks for selecting default files.
- *Why is it wrong*: Long methods are difficult to read and maintain. This also violates the Separation of Concerns principle, and the repeated logic leads to unnecessary duplication.

- *Possible solution:* Refactor the run method by splitting it into smaller methods with distinct tasks. Move the file reading logic into appropriate classes that handle those responsibilities.
- *Name of code smell:* Bloaters – Large Class, Dispensables – Duplicate Code

1.2.3 Code smells

- *Where:* The setCurrentSpools method in the StandardFM class.
- *What is wrong*: The setCurrentSpools method is unnecessary in the StandardFM class because its functionality is not relevant there. The StandardFM class only handles one color and does not require this method, which is only used in the MultiColor subclass.
- *Why is it wrong:* The presence of the method in the StandardFM class increases its complexity unnecessarily and this behavior is only relevant to the MultiColor class.
- *Possible solution*: Move the setCurrentSpools method to the MultiColor class where it is needed and remove it from the StandardFM class.
- *Name of code smell:* Unnecessary Method

1. 2.4 Code smells

- *Where:* The getSpoolByID method in PrinterManager, the CalculatePrintTime method in Printer, and the stringInput method in the Main class.
- *What is wrong:* These methods are not called anywhere in the codebase, making them redundant.
- *Why is it wrong:* Unused methods increase the complexity of the code, making the codebase harder to understand and maintain. Leaving dead code in the system can also lead to confusion about its purpose.
- *Possible solution:* Remove the getSpoolByID, CalculatePrintTime, and stringInput methods to simplify the codebase and eliminate unnecessary clutter.
- *Name of code smell:* Dead Code – Unused Methods

1.2.5 Code smells

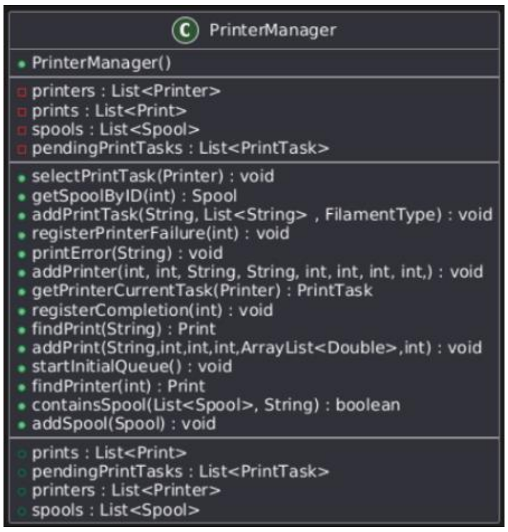
- *Where:* The reduceLength method in the Spool class
- *What is wrong*: The reduceLength method decreases the spool length twice, which can result in negative values.
- *Why is it wrong*: If the length of the spool is already zero or less, further reductions can lead to incorrect negative values. This logic is flawed and could cause unexpected behavior.
- *Possible solution:* Rewrite the reduceLength method so that the length is reduced only once, and only if it is greater than zero. Ensure there is a proper check to prevent further reductions when the length is already zero.
- *Name of code smell:* Bad method configuration

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.

The problem of the structure can be first seen in the **organization of the packages**, since there is only the model's class with many classes, this should be **further divided for better readability and managemen**t. With the current code example, it would be best if the Printers classes were separated into a package (i.e printer) so it can be easier to read.
Besides this structural problem, after closer code inspection one of the main issues becomes clear and that is the **tedious relationship between all the classes.** Down below we mark the bigger structural problems in more detail.

1- The problem in the PrintManager class, which has too many attributes and responsibilities, such as managing print, printers, spools, and pending Tasks. This is a problem because a class with too many responsibilities is harder to understand, maintain and violates the Separation of Concerns principle by handling unrelated tasks.
This can be a general Problem for this class because it is managing all the application in this case. It should be divided into other classes, making a Spool manager, a printer's manager and pending tasks class, for better separation of concerns.



```
PrinterManager
PrinterManager()
printers : List<Printer>
prints : List<Print>
spools : List<Spool>
pendingPrintTasks : List<PrintTask>
selectPrintTask(Printer) : void
getSpoolByID(int) : Spool
addPrintTask(String, List<String> , FilamentType) : void
registerPrinterFailure(int) : void
printError(String) : void
addPrinter(int, int, String, String, int, int, int, int,) : void
getPrinterCurrentTask(Printer) : PrintTask
registerCompletion(int) : void
findPrint(String) : Print
addPrint(String,int,int,int,ArrayList<Double>,int) : void
startInitialQueue() : void
findPrinter(int) : Print
containsSpool(List<Spool>, String) : boolean
addSpool(Spool) : void
prints : List<Print>
pendingPrintTasks : List<PrintTask>
printers : List<Printer>
spools : List<Spool>
```

Figure 7: Printer Manager Class

2- The Main class contains methods like readPrintsFromFile, readPrintersFromFile, and readSpoolFromFile that do not belong there. These methods should be in their appropriate classes (Prints, Printers, Spool) or a helper class, as keeping them in the Main class violates the Single Responsibility Principle. Besides this method there are other methods managing Printers (), Spools (show Spools), all these methods should either be in their respective classes, or in a manager/helper class. The responsibility of the class Main should be mainly for user interaction, hence, the code now present in the class does much more than just let the user interact. To make it better, a menu and user input would be the only concerns/methods in this class.

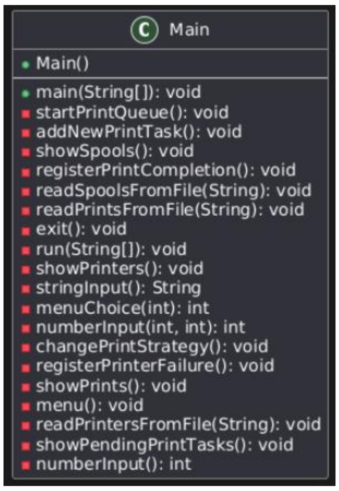

Main
Main()
main(String[]): void
startPrintQueue(): void
addNewPrintTask(): void
showSpools(): void
registerPrintCompletion(): void
readSpoolsFromFile(String): void
readPrintsFromFile(String): void
exit(): void
run(String[]): void
showPrinters(): void
stringInput(): String
menuChoice(int): int
numberInput(int, int): int
changePrintStrategy(): void
registerPrinterFailure(): void
showPrints(): void
menu(): void
readPrintersFromFile(String): void
showPendingPrintTasks(): void
numberInput(): int

Figure 8: Main Class

3- HousedPrinted class is unnecessary as it is an exact duplicate of StandardFM. Duplicate classes increase redundancy and maintenance, while also violating the DRY (Don't Repeat Yourself) principle. This would be better solved if StandardFM had an attribute (Boolean for example) which shows the state (whether it is housed printer or not) of the Printer

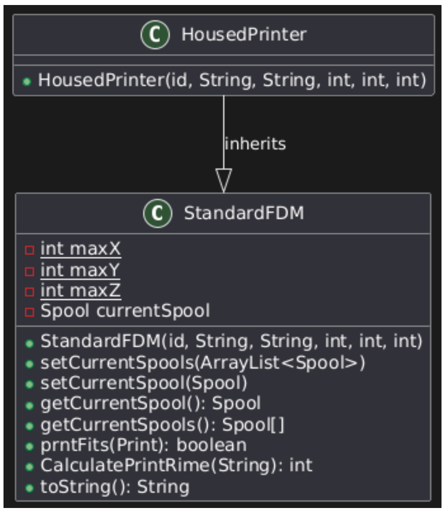

HousedPrinter
HousedPrinter(id, String, String, int, int, int)
inherits
StandardFDM
int maxX
int maxY
int maxZ
Spool currentSpool
StandardFDM(id, String, String, int, int, int)
setCurrentSpools(ArrayList<Spool>)
setCurrentSpool(Spool)
getCurrentSpool(): Spool
getCurrentSpools(): Spool[]
prntFits(Print): boolean
CalculatePrintRime(String): int
toString(): String

Figure 9: HousedPrinter and StandarFDM

4- PrintTask class at this moment is responsible for several actions, violating the Single Responsibility Principle (SRP). The tasks it does are: print details, colors, and filament type. This would be better resolved if the class was separated into more classes, or if the methods would go to a class that already deals with those responsibilities.

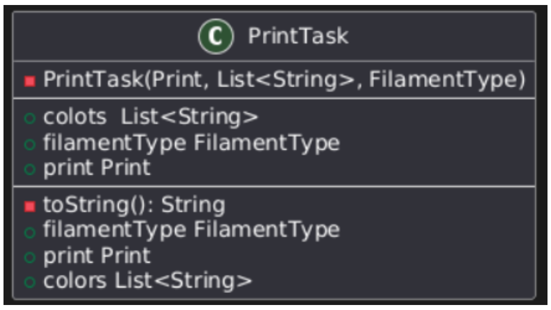
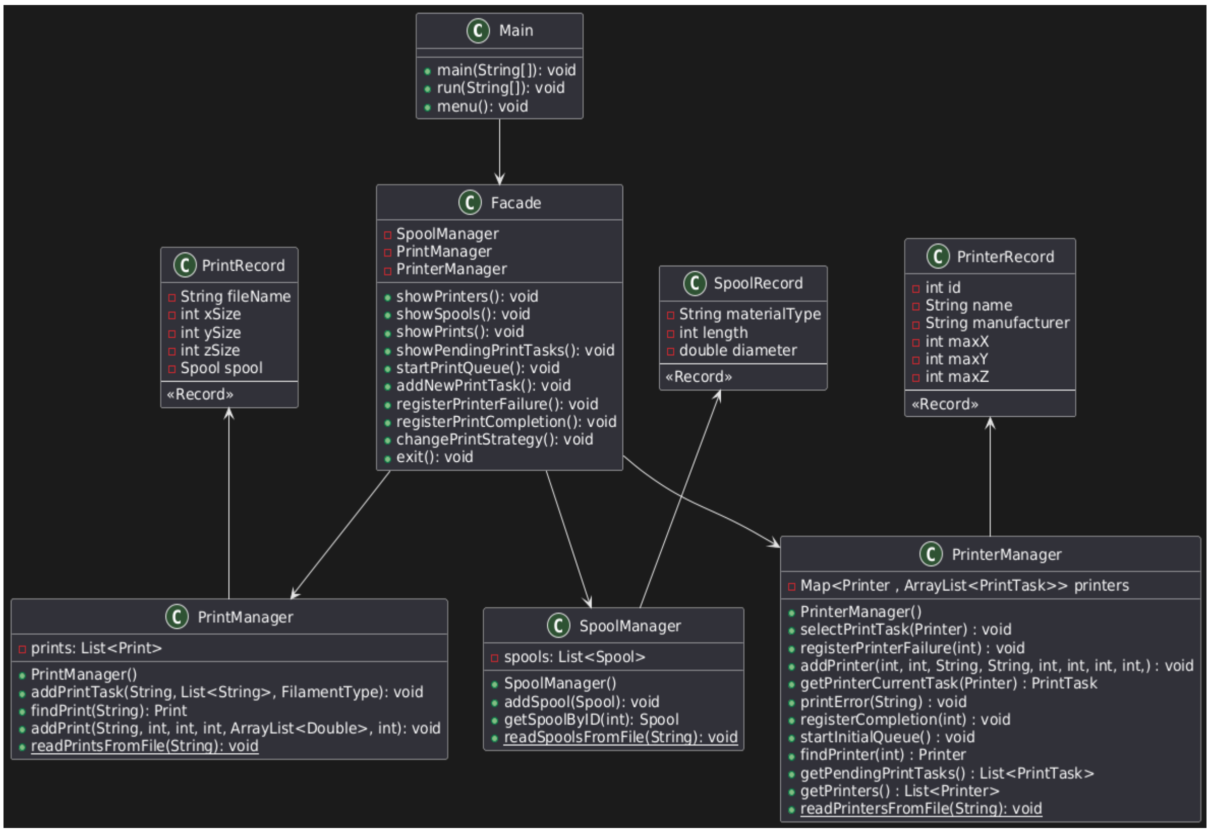

PrintTask
PrintTask(Print, List<String>, FilamentType)
colots List<String>
filamentType FilamentType
print Print
toString(): String
filamentType FilamentType
print Print
colors List<String>

Figure 10: PrintTask Clas

2. Design

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.



Main
main(String[]): void
run(String[]): void
menu(): void
Facade
SpoolManager
PrintManager
PrinterManager
showPrinters(): void
showSpools(): void
showPrints(): void
showPendingPrintTasks(): void
startPrintQueue(): void
addNewPrintTask(): void
registerPrinterFailure(): void
registerPrintCompletion(): void
changePrintStrategy(): void
exit(): void
PrintRecord
String fileName
int xSize
int ySize
int zSize
Spool spool
«Record»
SpoolRecord
String materialType
int length
double diameter
«Record»
PrinterRecord
int id
String name
String manufacturer
int maxX
int maxY
int maxZ
«Record»
PrintManager
prints: List<Print>
PrintManager()
addPrintTask(String, List<String>, FilamentType): void
findPrint(String): Print
addPrint(String, int, int, int, ArrayList<Double>, int): void
readPrintsFromFile(String): void
SpoolManager
spools: List<Spool>
SpoolManager()
addSpool(Spool): void
getSpoolByID(int): Spool
readSpoolsFromFile(String): void
PrinterManager
Map<Printer , ArrayList<PrintTask>> printers
PrinterManager()
selectPrintTask(Printer) : void
registerPrinterFailure(int) : void
addPrinter(int, int, String, String, int, int, int, int,) : void
getPrinterCurrentTask(Printer) : PrintTask
printError(String) : void
registerCompletion(int) : void
startInitialQueue() : void
findPrinter(int) : Printer
getPendingPrintTasks() : List<PrintTask>
getPrinters() : List<Printer>
readPrintersFromFile(String): void

Figure 11: New Main and Facade Diagram and their Relations

The updated system uses the **Single Responsibility Principle (SRP)** and the **Facade Design Pattern** to make the code simpler and easier to manage. The Facade class acts as a middleman between the Main class and the subsystems, so Main only handles basic application flow and user input. Tasks are split into three focused parts: PrinterManager for managing printers, PrintManager for handling print tasks, and SpoolManager for managing spools. This setup keeps responsibilities clear, reduces complexity, and makes it easier to add new features or fix bugs without breaking other parts of the system.
This structure ensures that each record class has a single responsibility and can be easily extended or modified. The Facade class interacts with these records through their respective managers (PrintManager, SpoolManager, and PrinterManager) to maintain a clean separation of concerns.

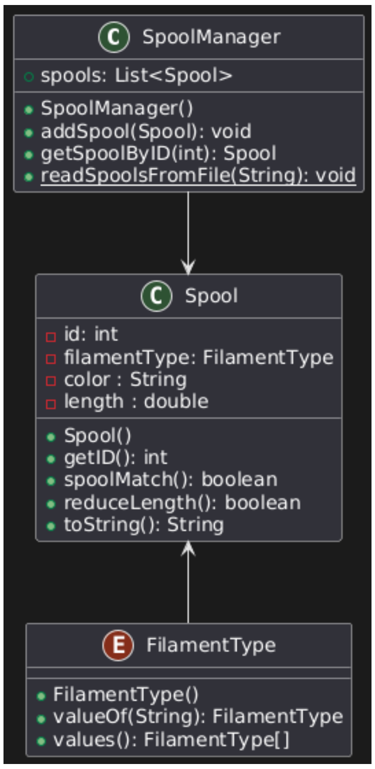


Figure 12: Spool Manager and its

The diagram adheres to the **Single Responsibility Principle (SRP)** as each class focuses on a specific task: SpoolManager manages spools by adding, retrieving, or reading them from a file, while the Spool class handles spool-specific details like id, filamentType, color, and operations such as spoolMatch() and reduceLength(). This keeps the responsibilities clear and reduces class interdependencies.
It also follows the **Open/Closed Principle (OCP)** by allowing the system to extend functionality without modifying existing classes. For instance, the FilamentType enum can support new filament types without changing the logic in Spool or SpoolManager. Lastly, the **Liskov Substitution Principle (LSP)** is observed as the SpoolManager interacts with Spool objects in a way that ensures substitutability, treating all Spool instances consistently through its methods like addSpool() and getSpoolByID().

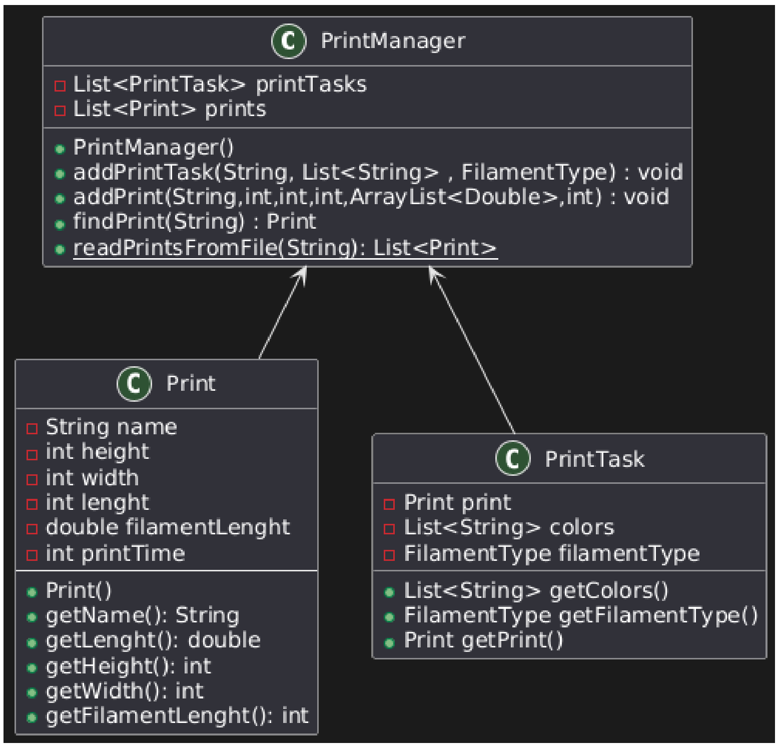


Figure 13: PrintManager and its Relations

In this diagram, the **Single Responsibility Principle (SRP)** is followed as each class has a specific role: PrintManager handles print-related tasks and management, Print defines the properties of a print (e.g., name, height), and PrintTask manages task-specific details like colors and filament type. This keeps responsibilities clear and organized.
The **Open/Closed Principle (OCP)** is evident as new functionalities (e.g., methods in PrintManager) can be added without modifying existing code. The **Liskov Substitution Principle (LSP)** is adhered to since PrintManager interacts with Print and PrintTask in a way that allows extensions or replacements without breaking functionality.

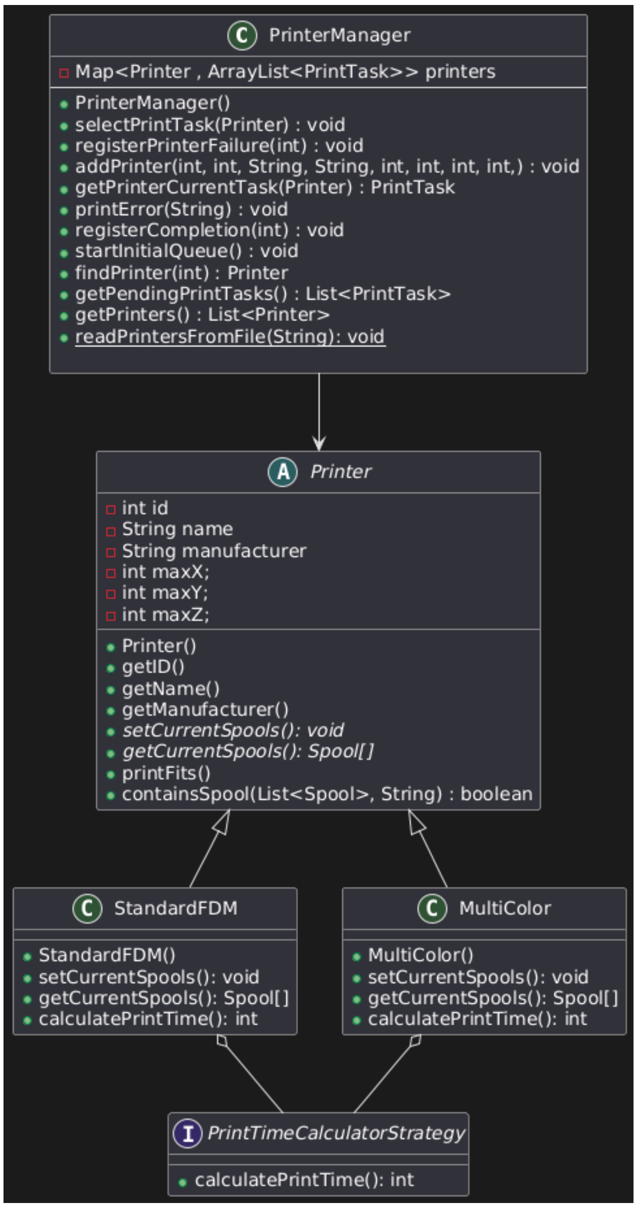

PrinterManager
Map<Printer , ArrayList<PrintTask>> printers
PrinterManager()
selectPrintTask(Printer) : void
registerPrinterFailure(int) : void
addPrinter(int, int, String, String, int, int, int, int,) : void
getPrinterCurrentTask(Printer) : PrintTask
printError(String) : void
registerCompletion(int) : void
startInitialQueue() : void
findPrinter(int) : Printer
getPendingPrintTasks() : List<PrintTask>
getPrinters() : List<Printer>
readPrintersFromFile(String): void
Printer
int id
String name
String manufacturer
int maxX;
int maxY;
int maxZ;
Printer()
getID()
getName()
getManufacturer()
setCurrentSpools(): void
getCurrentSpools(): Spool[]
printFits()
containsSpool(List<Spool>, String) : boolean
StandardFDM
StandardFDM()
setCurrentSpools(): void
getCurrentSpools(): Spool[]
calculatePrintTime(): int
MultiColor
MultiColor()
setCurrentSpools(): void
getCurrentSpools(): Spool[]
calculatePrintTime(): int
PrintTimeCalculatorStrategy
calculatePrintTime(): int

Figure 14: PrinterManager and its Relations

The diagram adheres to the **Single Responsibility Principle (SRP)** by assigning specific roles to each class. For instance, PrinterManager handles printer operations like adding printers and managing queues, while Printer represents individual printers with attributes like ID and dimensions. Subclasses like StandardFDM and MultiColor manage specific printer types with distinct spool and print time calculations.
It follows the **Open/Closed Principle (OCP)** by allowing new printer types to inherit from Printer or new strategies to implement PrintTimeCalculatorStrategy without altering existing classes. The design supports the Liskov Substitution Principle (LSP) by enabling PrinterManager to work seamlessly with any Printer subclass or strategy. Additionally, **Interface Segregation Principle (ISP)** is reflected in PrintTimeCalculatorStrategy, isolating calculation functionality and allowing new strategies to integrate without affecting other components.

2.2 Design patterns

2.2.1 Façade

Adding the façade to the code will not necessarily seem required right away, but once it is implemented, it does a great job cleaning the main class and applying the separation of concerns.

Firstly, the change which is noticeable is adding all the menuOptions in Façade and keeping Main only for user interaction. This was a task to us because even after cleaning up the methods and putting them in the dedicated managers, there were still lots of methods

Secondly, by keeping everything in the façade, we ensure that the clients only have access to the appropriate functionality, keeping the access centralized and secure. For example, as disscues in class, If there are multiple developers working on the same code and someone by 'accident' or 'mistake' edits the list of printers, that will be applied to the whole system. They have access to mess with everything in the code. Adding the facade and the records ensures we keep to good coding principles and no such 'accidents' can happen.

2.2.2 Strategy Pattern

When creating the class diagrams using the given code, having the calculatePrintTime as an abstract method in Printer didn't seem like a bad idea or even code smell. The only concern with this was that Printer now handles something it isn't supposed to , calculating time. After learning in class about the Strategy Pattern and its implications, having this method in Printer seemed less logical. That is why, with this we not only gained to comply to the Single responsibility principle we also adhere to the Open/Closed.

And right now, if for example there is a normal printer added, which does not extend Printer but it has its own separate class, than implementing this interface would be easy and no changes to any of the classes would be needed.

This proves that this strategy Pattern improved the code in way that it makes it easier to upgrade and maintain (everything has its own concern).

2.2.3 Factory Method

Adding the Factory Method was a bit strange, and it took us some time to understand and discuss whether we should add it to the code. However, after considering the benefits, we agreed it would improve the structure, and once implemented, the code looked much cleaner and more organized. Using the PrinterFactory has really helped us separate concerns and make the system easier to manage.

As we added the PrinterFactory, the main change is that instead of creating printers directly in the PrinterManager, we now use the PrinterFactory class.

This keeps the printer creation process separate, which makes the code more modular and easier to extend. If we want to add a new type of printer in the future, we only need to add a new argument in the factory class, and the rest of the system will remain unchanged. This reduces the risk of bugs and makes the code easier to update and maintain.

2.2.4 Observer

Adding the Observer Pattern to our system was a significant step toward improving communication between components. Initially, managing the status updates of printers or print jobs was tedious and error-prone. Now, with the addition of the PrinterObserver interface and the notifyObservers method in the PrinterManager, we can easily notify all components about changes, such as when a printer's status changes.

The main change was that, instead of each component constantly checking the status of printers, we now have observers that are notified whenever there is an update. The PrinterManager acts as the subject, while the PrinterObserver objects (such as PrinterManager itself or other components) can react to changes by implementing the update method.

This separation of concerns makes the system more modular and flexible. We can now add new observers without modifying the core logic of the printers or the PrinterManager. If we need to add a new type of observer (e.g., a PrinterLogger), we simply implement the PrinterObserver interface and register it. This makes the system more extensible and easier to maintain as new requirements arise.

2.3 Global overview

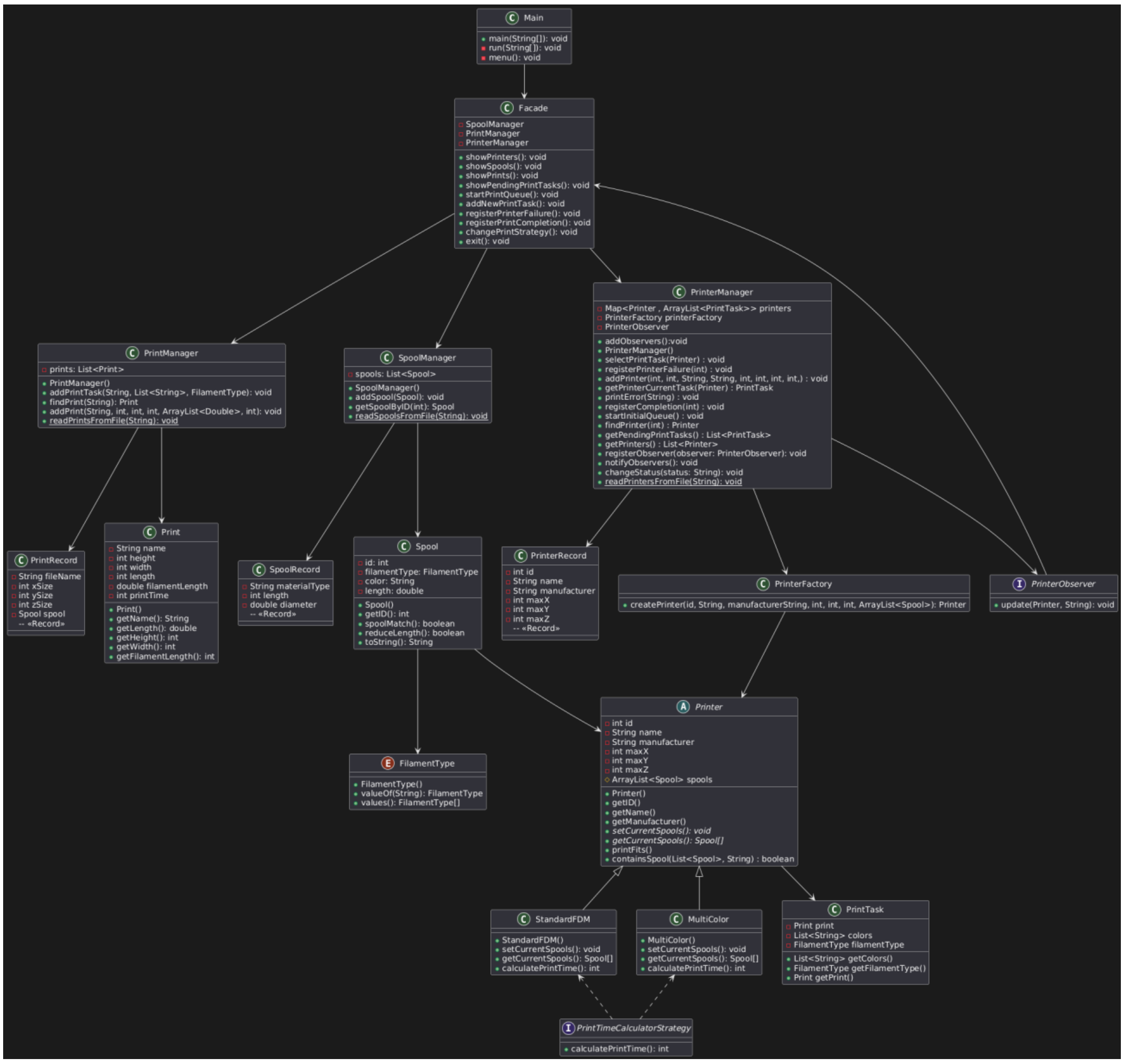


Figure 15: Global Class Diagram with new Design patterns

The diagram is the overview of how the application would look once refactored, this is the initial design we came up with. It looks much less tedious than the class diagram before. To achieve this, we first introduced 3 new classes (the façade and the 2 manager classes) to make the code more readable and clear and we eliminated one class (The HousedPrinter class).

We have tried to stick to the SOLID principles and implemented some design patterns, so that it sticks to the conventions mentioned in the SOLID principles. The 3 new classes were the management classes, for Printer, Print and Spool objects. Furthermore, these classes deal with their respective responsibilities – managing the applicable object (Printer, Print, Spool). We have moved some of the methods from PrinterManager and Main class to their respective new classes. We added some more methods and are planning on improving or fixing the already given logics.

To ensure that we implemented the Façade strategy (more explanation is given in 2.2.1), we created new Record classes(can be seen in the diagram, marked as Record in the class itself), PlantUML doesn't have a separate logo for this, at it is not a feature in all programming languages so we marked it like this <<Record>>. In the classes used (i.e. Printer, Print, Spool), all the attributes are made private so that they can't be accessed from outside the class, the getters are only accessible in the managers, ensuring no data can be leaked, or modified outside in Main. For this we have the façade which acts in a way as a layer between the user interaction application and the code.

Moreover, we also introduced the strategy pattern with which PrintTimeCalculationPattern is used for calculating the time it takes to print. This makes the Printer class closed to modification but open to extending, hence adhering to the open closed principle (for more read 2.2.2). We also made sure that filamentType is only used at one place (the spool), instead of having associations with other classes.

Overall, we hope that with this diagram we have improved the structure and have explained it well enough so that we can start implementing the code.

3. Implementation

3.1 Implement your design

Implementation of our design initially meant refractor of the given code. We started off by splitting up the work and just doing some basic refactoring. We began by introducing the manager classes and moving the methods from the other classes (like the Main and Printer manager) to their respective new classes. Doing this wasn't such a big issue and it was exactly as we discussed and drew in the class diagram (see *figure 15).*

One thing we did not predict was the connection between the managers, for example with our code implementation we realized that the list of Spools and Print Task we read from the file(the reading methods are placed in the respective managers)were needed in the Printer Manager class as so that the when assigning Print Task to the Printer can be done correctly. That is why when coding we added the same instance of the Spool and Print manager to the Printer manager.

This added more complexity to the system as now in Printer manager we have access to the spools, however we were able to make the system function as we were intending it to.

While we made many betterments to the system, like eliminating the code smells (simplifying methods, splitting up large methods) we also realized that we would need to do some changes to how we planned to implement the system. This change included making an association relationship between Printer and Spool manager, so that Printer has access to the list of Spools (we are aware that there might be a better solution but implementing this meant using more time and bigger changes which is not of help).

3.2 Extension of the system

Extension set 1

In this expansion, we integrated a dashboard into the system using the Observer pattern. The Observer pattern was implemented by adding an Observer interface with an update method, which is used to notify observers whenever specific events occur. A Dashboard class was created as the concrete observer, designed to track and display the number of spools changed, the total number of prints completed and failures.

To make the dashboard accessible, a new option was added to the system menu. Selecting this option displays the current dashboard data in a simple, text-based format.
The system was also updated to notify the Dashboard whenever spools are changed or print jobs are completed or failed. Initially, there were some challenges in ensuring the spool change data was correctly updated in the dashboard, but these issues were resolved.

Overall, the implementation was straightforward once the Observer pattern's structure was clear, and the system now offers an extensible way to track data.

Extension set 2

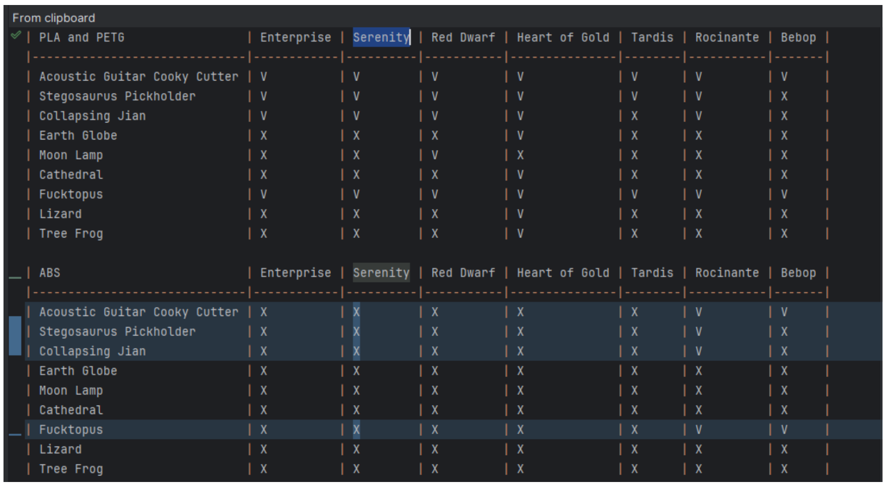
In the second expansion, we updated the system to handle both CSV and JSON files based on the file extension the user provides. The code now checks the file type, selects the correct class, and reads the data.

To do this, I added one interface and two classes (CSVAdapterReader and JSONAdapterReader) to keep things organized and flexible.
We also added a new printer with multicolor capabilities, categorized as type "4." This makes the system more useful and ready for future updates. While it was tricky at first, this setup works well and keeps things simple.

Finally, the second strategy was implemented which optimizes the spool usage. What this strategy ensures is that the spools with minimal lengths get used up first.

3.3 Testing

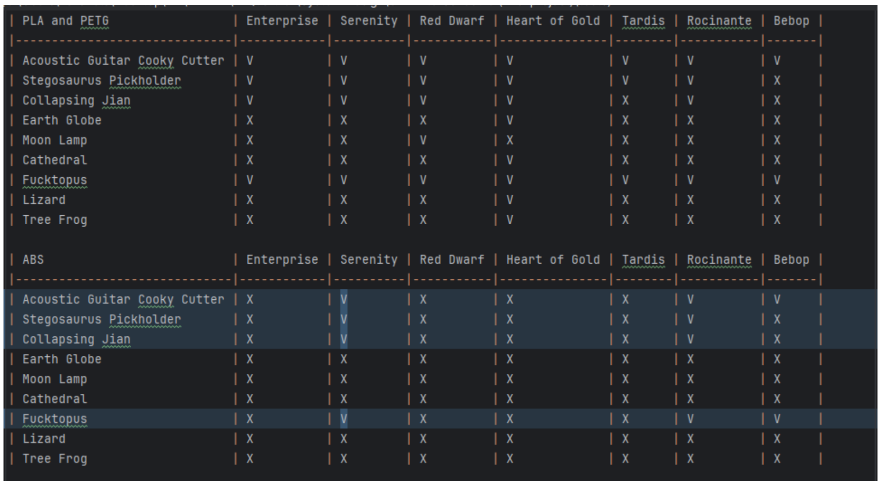
The way, we decided to test our application by creating a test class that will generate an MD file with 1 table for each Filament Type and see if the print can be printed in the selected Printer. At first, we had so many problems testing our application that we had to change the logic of the application because we missed a big Check while assigning the task to a printer. To get our tables just run our test class and a document will be generated.



PLA and PETG	Enterprise	Serenity	Red Dwarf	Heart of Gold	Tardis	Rocinante	Bebop
Acoustic Guitar Cooky Cutter	V	V	V	V	V	V	V
Stegosaurus Pickholder	V	V	V	V	V	V	X
Collapsing Jian	V	V	V	V	X	V	X
Earth Globe	X	X	X	V	X	X	X
Moon Lamp	X	X	V	X	X	X	X
Cathedral	X	X	X	V	X	X	X
Fucktopus	V	V	V	V	V	V	V
Lizard	X	X	X	V	X	X	X
Tree Frog	X	X	X	V	X	X	X

ABS	Enterprise	Serenity	Red Dwarf	Heart of Gold	Tardis	Rocinante	Bebop
Acoustic Guitar Cooky Cutter	X	X	X	X	X	V	V
Stegosaurus Pickholder	X	X	X	X	X	V	X
Collapsing Jian	X	X	X	X	X	V	X
Earth Globe	X	X	X	X	X	X	X
Moon Lamp	X	X	X	X	X	X	X
Cathedral	X	X	X	X	X	X	X
Fucktopus	X	X	X	X	X	V	V
Lizard	X	X	X	X	X	X	X
Tree Frog	X	X	X	X	X	X	X

Figure 16: Table from Tested Results



```
| PLA and PETG                | Enterprise | Serenity | Red Dwarf | Heart of Gold | Tardis | Rocinante | Bebop |
|-----------------------------|------------|----------|-----------|---------------|--------|-----------|-------|
| Acoustic Guitar Cooky Cutter | V         | V        | V         | V             | V      | V         | V     |
| Stegosaurus Pickholder      | V          | V        | V         | V             | V      | V         | X     |
| Collapsing Jian             | V          | V        | V         | V             | X      | V         | X     |
| Earth Globe                 | X          | X        | X         | V             | X      | X         | X     |
| Moon Lamp                   | X          | X        | V         | X             | X      | X         | X     |
| Cathedral                   | X          | X        | X         | V             | X      | X         | X     |
| Fucktopus                   | V          | V        | V         | V             | V      | V         | V     |
| Lizard                      | X          | X        | X         | V             | X      | X         | X     |
| Tree Frog                   | X          | X        | X         | V             | X      | X         | X     |

| ABS                         | Enterprise | Serenity | Red Dwarf | Heart of Gold | Tardis | Rocinante | Bebop |
|-----------------------------|------------|----------|-----------|---------------|--------|-----------|-------|
| Acoustic Guitar Cooky Cutter | X         | V        | X         | X             | X      | V         | V     |
| Stegosaurus Pickholder      | X          | V        | X         | X             | X      | V         | X     |
| Collapsing Jian             | X          | V        | X         | X             | X      | V         | X     |
| Earth Globe                 | X          | X        | X         | X             | X      | X         | X     |
| Moon Lamp                   | X          | X        | X         | X             | X      | X         | X     |
| Cathedral                   | X          | X        | X         | X             | X      | X         | X     |
| Fucktopus                   | X          | V        | X         | X             | X      | V         | V     |
| Lizard                      | X          | X        | X         | X             | X      | X         | X     |
| Tree Frog                   | X          | X        | X         | X             | X      | X         | X     |
```

Figure 17: Table given by teachers

Also we think we found an error in the table provided by us in the table the Printer Serenity is type 1 meaning it is not Housed but, in the table, given to us the Printer Serenity says it can print ABS what is impossible cause the only types that can print ABS are StandardFDM type 2 or MultiColor type 4 and the Printer Serenity is StandardFDM type 1 meaning it can't print ABS that's why we think we might have found an error.

4. Conclusion

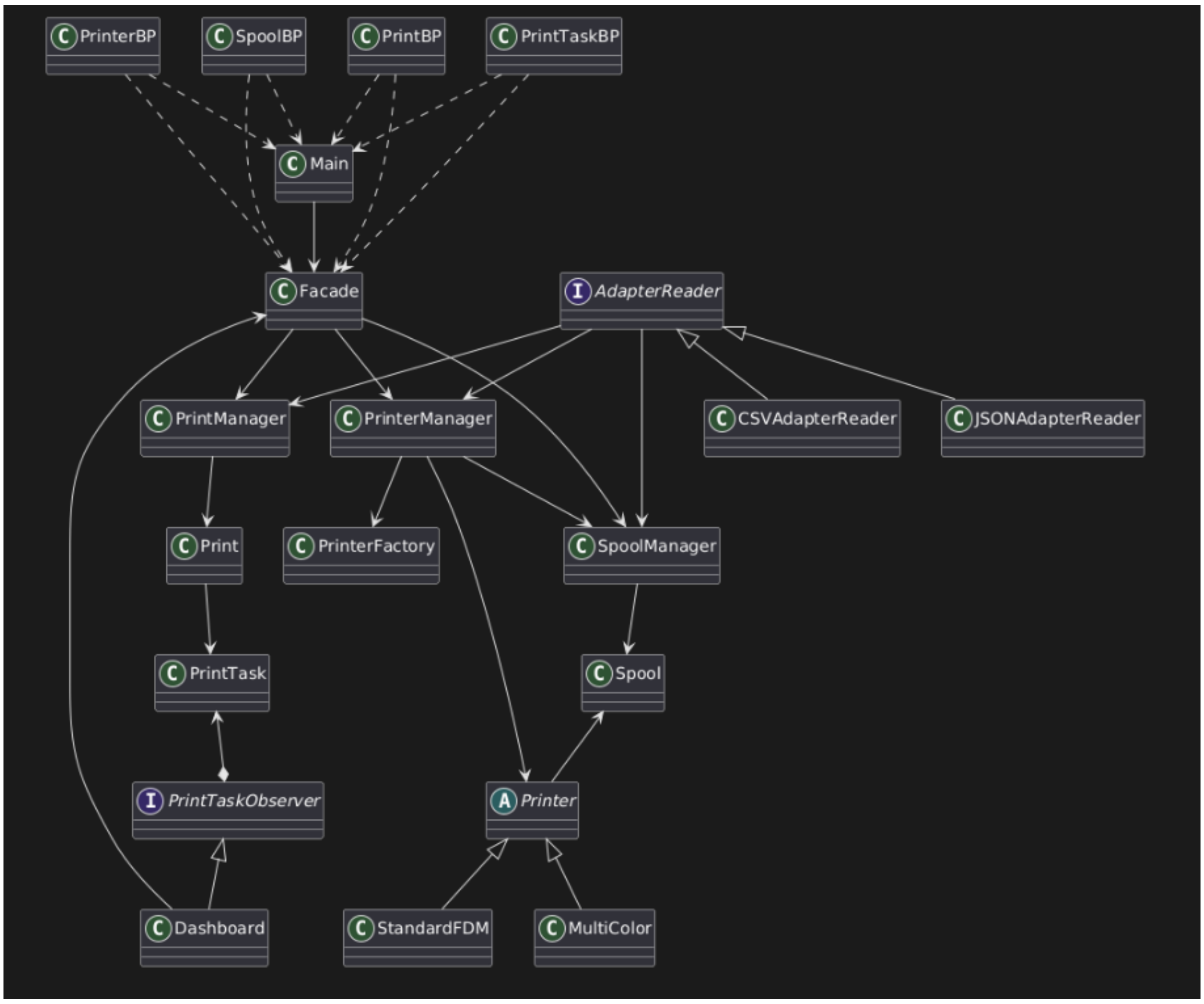


Figure 18: Global Class Diagram after refracturing and 2 expansions

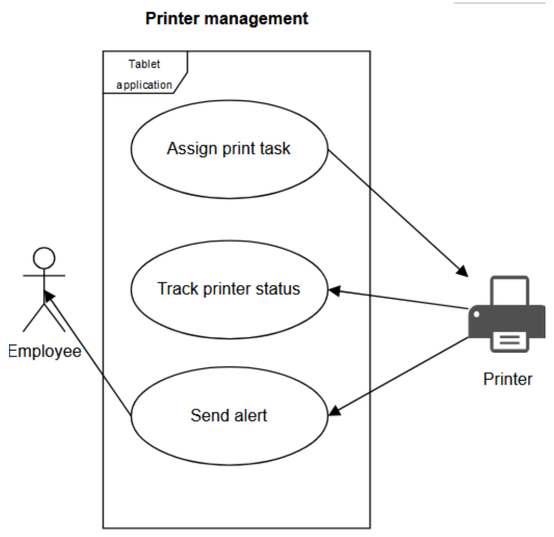
Concluding our 1 Part of this document this is the result of our Class diagram after the refracturing and expansion's. We decided to make the diagram easy simple and readable to get an overall overview since we already talked about how we would fix it and what we implemented before.

Part 2: Software architecture

5. Software architecture

Use case diagrams

The following use case diagram describe the system in an easy and understandable manner



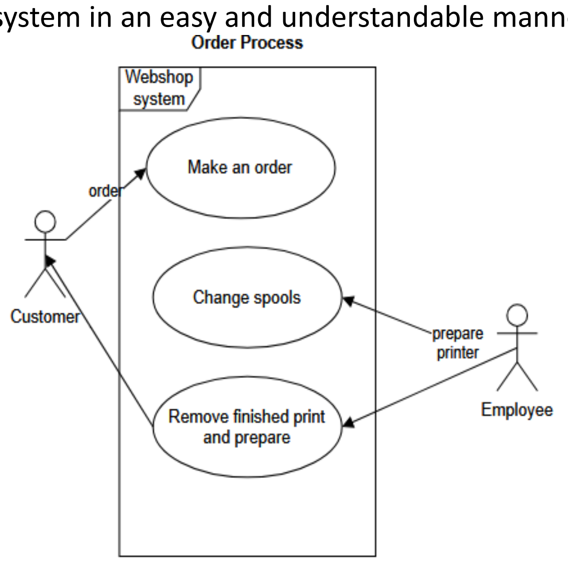


Figure 1: Use case diagram for the ordering process Figure 2: Use case diagram for the ordering process

This diagram shown in Figure 1 focuses on the interaction between the customer, webshop system, and employee to ensure smooth order process. It includes:

- Customer placing an order via the webshop.
- Webshop forwarding the order to the appropriate print farm.
- Printing starts
- Updating the webshop about the print's readiness for shipping or pickup.

The diagram shown in Figure 2 highlights the system's functionality in managing the printers effectively. It includes:

- Assigning print jobs to available printers.
- Monitoring printer status and sending notifications for completed prints.
- Allowing employees to register a printer as ready.
- Tracking printer usage statistics to inform the owner about performance and material consumption.

Sketch of the System

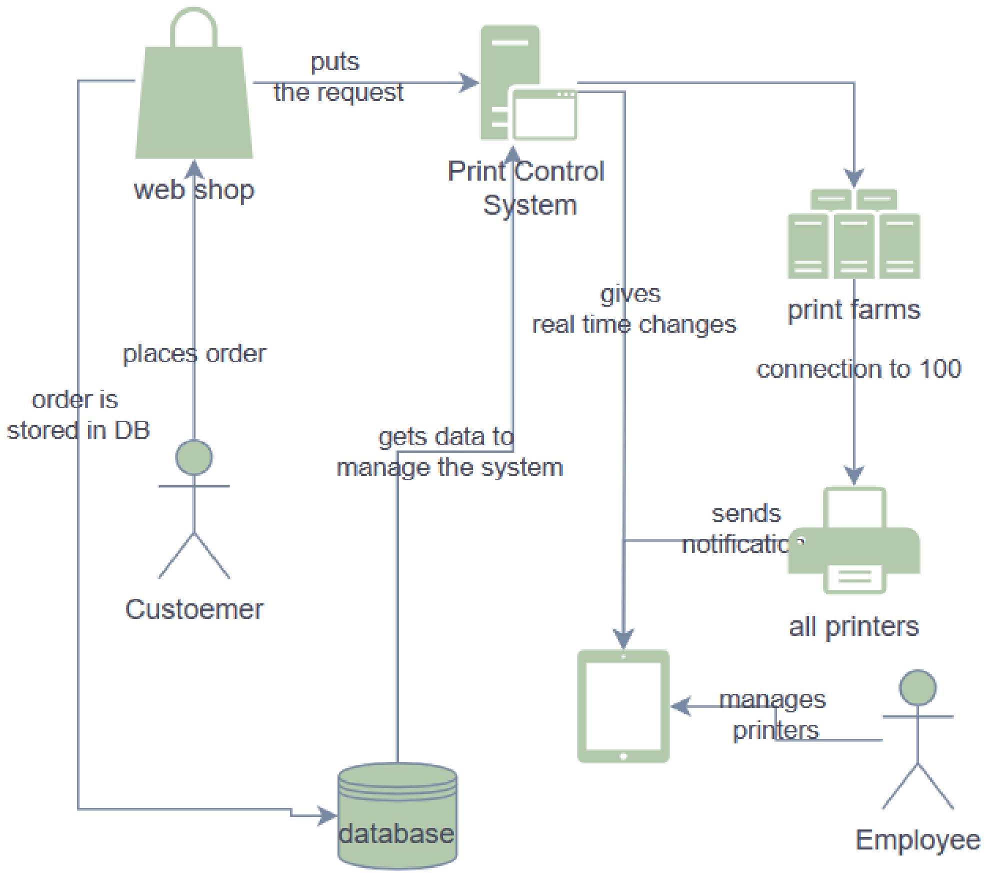


Figure 3 Overall sketch of the system

Webshop System (presented with the gray bag):

- The central hub where customers place orders.
- Orders are forwarded to the Print Control System and stored in the database.

Print Control System:

- Handles print jobs, queues them for printers, and tracks statuses.
- Communicates with the Print Farms and updates the Webshop System.

Print Farms:

- Consist of print farm computers connected to 100 printers.
- Employees use tablets to manage printer status.

Tablets:

- Provide employees with real-time updates.
- Allow employees to mark printers as ready and manage spool changes.

Database:

- Stores critical data such as order details, job statuses, and printer performance.

Component and Deployment Diagram

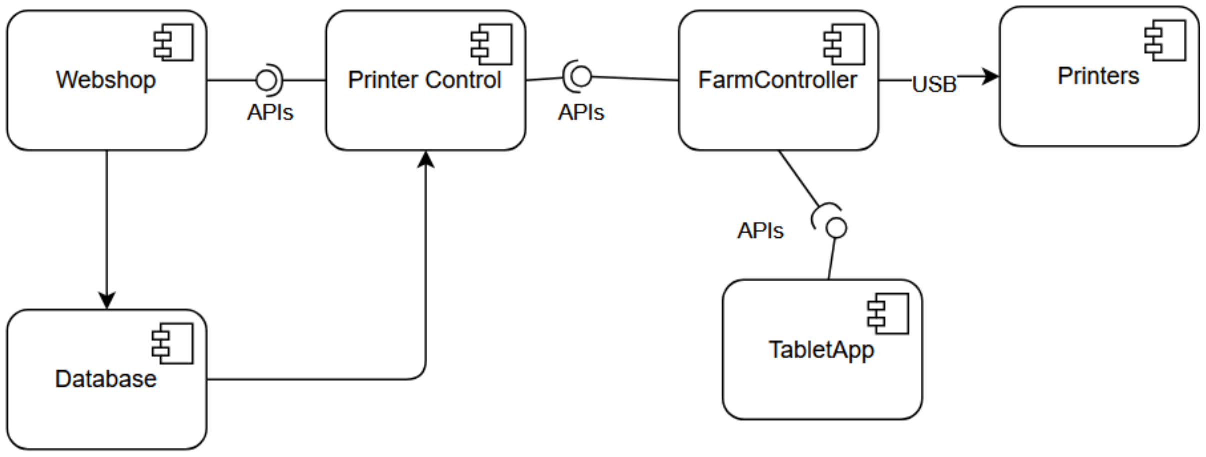


Figure 4: Component diagram

The diagram clearly visualizes how the components in this whole system communicate with each other and what they use to communicate. Beginning with the WebShop, which has connection to the Database (to retrieve and write data), this is an implementation choice. For example, the Webshop could only communicate with the Printer Control, however that makes the job of the Printer Control more complicated and it adds extra things to manage violating the Single Responsibility principle.

Additionally, there is the Printer Control which has a connection to the Database to write (print finished, ready to ship, delivered/picked up) and retrieve data (Print data). The other connection that Printer Control makes are the one with the Farm Controllers. The Farm Controller communicates with all Printers directing Prints to the appropriate Printers (similar to something we have already build). The Farm Controller is also responsible for communicating with the Tablet App via API.

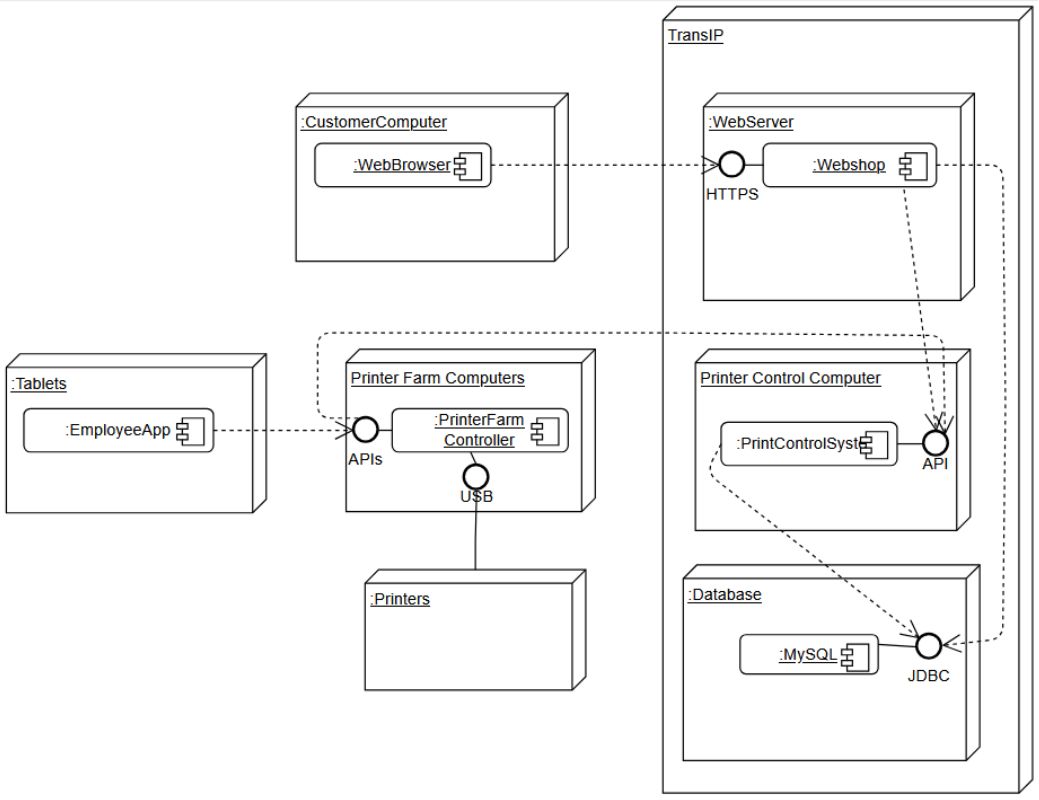


Figure 5: Deployment diagram

The deployment diagram shown in Figure 5 depicts the big picture of the system and how the communication in the system happens.

As it can be seen from the diagram there is the big TransIP VPS which hosts the three big hardware components of the system. It hosts the Web Server, the Printer Control Computer/s, and the Database. This means that the communication inside this component is secured by the TransIP connection.

To elaborate in more detail for each one of them:

- The web server hosts the webshop which communicated with the database through Java Database Connectivity (JDBC)
- The PrintControlSysem also uses the database and communicates with it through JDBC
- Finally, the Webshop uses API communication with the Print Control System

Outside of the VPS is the connection from the client computer, who uses web browser and runs the webshop app via HTTPS protocol. Another connection which happens outside the VPS is to the Printer Farm Controller which happens also via API. Finally, there is the connection between Printers and the Print Farm System which happens via USB connections,

Activity diagram for requesting a 3D print

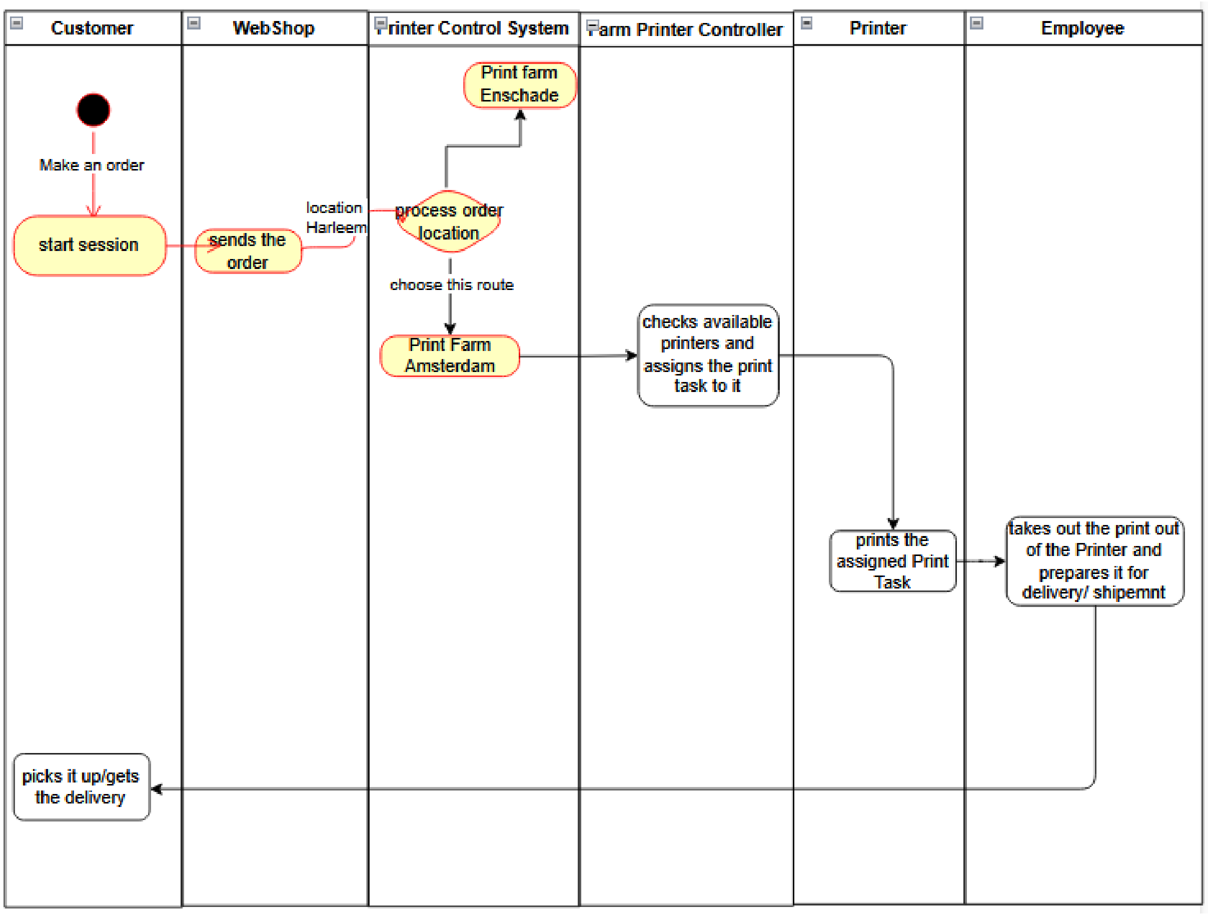


Figure 6: Activity diagram for requesting a 3D print

This activity diagram represents the order of a 3D print and its full workflow. It begins by a customer placing an order on the web shop. They initially put their order details, including type of print, color, material and a preference for pickup or shipping.

The order than goes through to the Print Control System which makes the decision to

Start:

- The customer places an order on the webshop.
- Enter order details (e.g., product, color, material, and pickup/shipping preference).

Print Job Queuing:

- The Print Control System assigns the job to the appropriate Printer Farm.
- The Printer Farm then directs it to the correct Printer.
- The job is added to the printer queue.

Printing Process:

- Printer starts the job.
- Print completion is monitored by the Farm Printer System.

Print Completion Notification:

- When the print is complete, the Farm Control System sends a notification to the employee's tablet.

Employee Actions:

- Employee removes the finished print from the printer.
- Employee marks the printer as ready for the next job using the tablet.

Packaging and Labeling:

- Employees package the print and label it according to the order details.
- The order is now ready for pickup or shipping.

Potential implementation of architecture patterns

The publish/subscribe architecture pattern appears to be a valid and efficient choice for the system. In this implementation:

- The Print Control System acts as the publisher, broadcasting print tasks.
- The Printer Farm Controllers function as subscribers, listening for tasks and processing them when capacity is available.

This approach shifts the responsibility of task distribution from the Print Control System to the Printer Farms, allowing them to subscribe to tasks dynamically based on their availability. This decentralizes task allocation, reducing the complexity and workload on the Print Control System. This makes the Print Control System faster, more efficient and easier to maintain, as now the system isn’t concerned with anything but assigning task.-

1. Scalability
- Easily add new Print Farm Computers or components without modifying existing systems.
- Handles increasing workloads by distributing messages efficiently.

2. Real-Time Notifications
- Subscribers receive updates as soon as new messages are published, ensuring timely operations.

3. Loose Coupling
   - 4. The Central Print Control System (publisher) and Print Farm Computers (subscribers) operate independently.

One downside of this method is that if a subscriber goes offline, the jobs it had would need to be redistributed.

Sequence diagrams

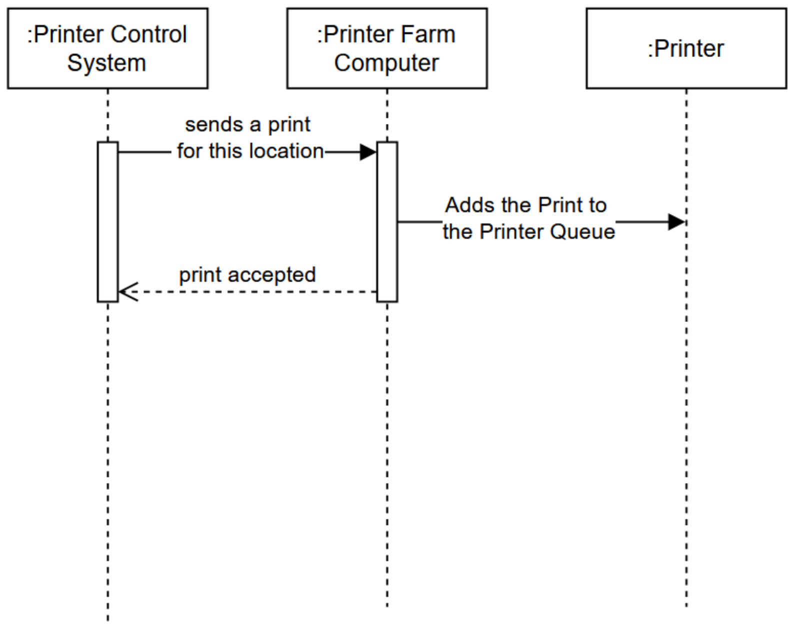


Figure 7: Sequence diagrams

Description for 1 sequence diagram - This sequence diagram shows the interaction between Printer Control System, Printer Farm Computer and Printer. The process begins with the Printer Control System sending a print request for a specific location to the Printer Farm Computer. The Printer Farm Computer then adds the print request to the printer queue. Once the request is queued successfully, the Printer Farm Computer confirms the acceptance of the print request back to the Printer Control System.

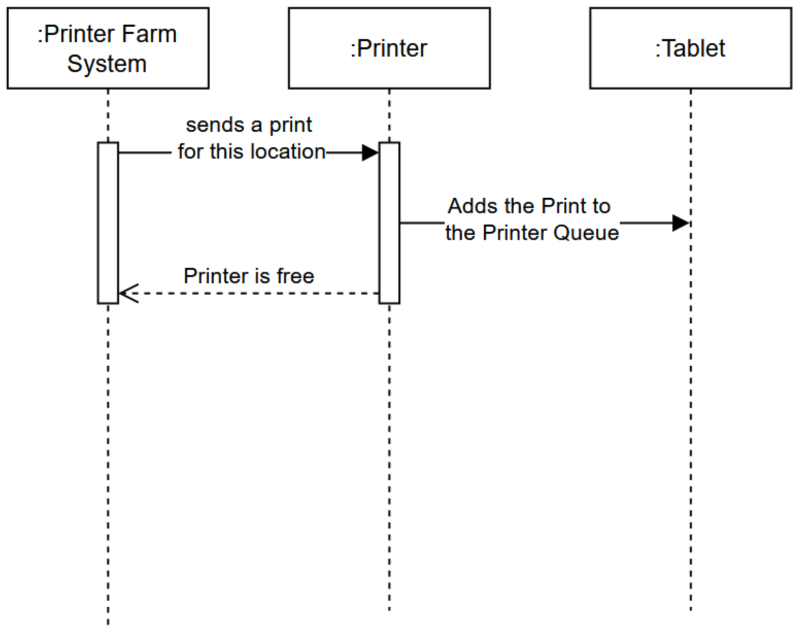

:Printer Farm System
:Printer
:Tablet
sends a print for this location
Adds the Print to the Printer Queue
Printer is free

Figure 8: Sequence diagrams

Description for 2 sequence diagram - This sequence diagram shows the interaction between the Printer Farm System, Printer, and Tablet. The Printer Farm System sends a print request for a specific location to the Printer. The Printer processes the request and adds it to the printer queue. Once the request is queued and the Printer is available, it notifies the Printer Farm System that it is free.

Alternatives not Implemented.

When designing the communication architecture between the Print Control System (PCS) and the print farms, we carefully considered various options, including using a message broker like RabbitMQ. Ultimately, we decided against this approach and opted for a simpler solution using APIs via a Virtual Private Server (VPS).

Why We Chose API/VPS

- **Simplicity**: Easier to implement and maintain compared to a message broker.
- **Cost-Effective**: No need for additional infrastructure, as the VPS handles all communication.
- **Sufficient for Needs**: Handles the current print request volume without overengineering.
- **Scalability**: Easily expandable for future growth.

Why We Did Not Choose Message Broker

- **Added Complexity**: Increases system setup and maintenance effort.
- **Additional Server**: Requires dedicated infrastructure, raising costs.
- **Failure Point**: Introduces another component that could fail.
- **Overkill**: The volume of print requests does not justify the use of a message broker.

Future Expansions

New Print Farms in Additional Locations:

- Adding new print farms in other cities because the architecture supports decentralization. The Print Control System can easily communicate with new farms via the message broker.

Customer Notification:

- The system could be expanded to send real-time notifications to customers (via email or SMS) about print status, including estimated completion times.

Automated Spool Management:

- Future integration with IoT-enabled spools could allow the system to automatically detect when a spool is running low and notify employees.

Convincing the reader

Our choices for the Printer Management System are rooted in practicality, cost-efficiency, and scalability. By leveraging API communication via a VPS, we ensure a simple yet powerful solution that meets the current needs of the system without overengineering. The design adheres to fundamental software principles, minimizes risks, and paves the way for future growth and innovation. This thoughtful approach ensures the system’s long-term reliability and adaptability.