

# Technical report

System Design

Rubén Gómez Hidalgo  
Carlos Guinovart i Galofre  
Course: 2023-2024  
Date: 25/03/2024



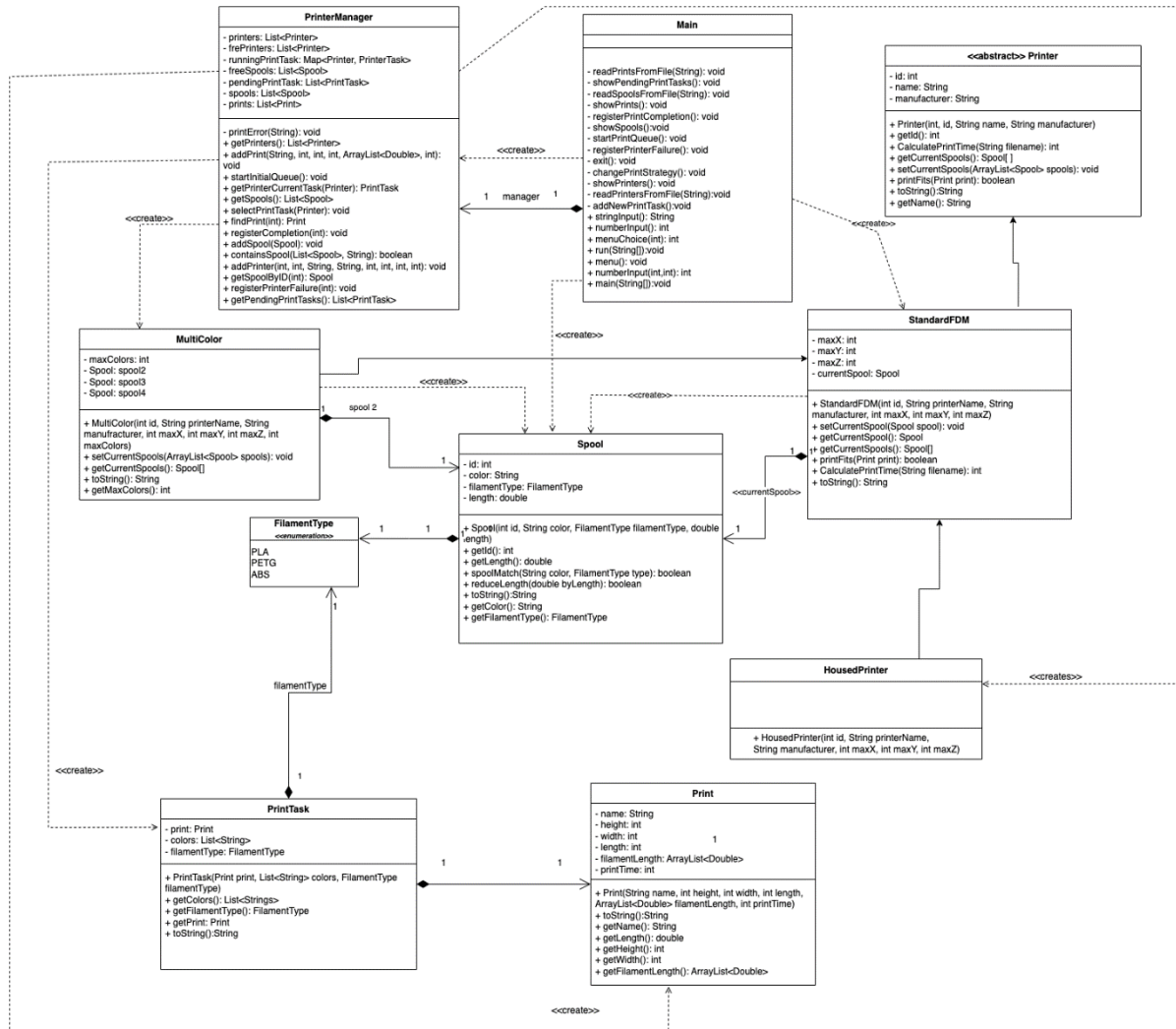
Week 1: Code analysis .....	3
1.1 Class diagram .....	3
1.2 Code smells .....	4
Week 2: Refactoring .....	5
2.1 Applying software principles.....	5
2.2 Facade pattern .....	7
Week 3: Design patterns .....	10
3.1 Strategy and singleton pattern.....	10
3.1.1 Strategy pattern .....	10
3.1.2 Singleton pattern.....	13
3.2 Adaptor and factory (TODO: Richard) .....	14
3.2.1 Adaptor pattern .....	14
3.2.2 factory pattern .....	16
Week 4: More design patterns.....	18
4.1 Observer pattern.....	18
Week 5: Expansion of the system .....	20
5.1 Testing .....	20
5.2 Conclusion.....	20
Week 6: Architecture .....	21
6.1 Architecture .....	21
6.2 Connection between components and activity diagrams .....	28
6.2.1 activity diagram.....	28
6.2.2 architectural patterns.....	28
Week 7 + 8 Proof of concept.....	30

# Part 1: Software modelling

## Week 1: Code analysis

### 1.1 Class diagram

- Create a class diagram from the 3dprintscheduler-main



- Add new Print Task:** Main Class, PrinterManager Class and Print()
- Register Printer Completion:** Main(), PrinterManager(), Printer() and PrinterTask()
- Register Printer Failure:** Main(), PrinterManager(), Printer() and PrintTask()
- Change printing style:** Main()
- Start Print Queue:** Main(), PrinterManager(), Printer(), PrintTask(), StandardFDM(), Spool()
- Show prints:** Main(), PrinterManager(), Print()

7. **Show printers:** Main(), PrinterManager(), Printer() and PrintTask()
8. **Show spools:** Main(), PrinterManager() and Spool()
9. **Show pending print tasks:** Main(), PrinterManager(), PrintTask()
10. Exit

## 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*

- 1) **Large Class:** This code smell can be found in PrinterManager class. The problem with this class is that it has too many responsibilities. The possible solution that could be given to solve this code smell is to separate functionalities to other classes.
- 2) **Long Method:** This code smell can be found in Main class. The problem in this case is that in this class it exists large methods like method "addNewPrintTask", "readPrintsFromFile", "readSpoolsFromFile" that these methods perform too many tasks. The possible solution could be to separate functionality into other private methods.
- 3) **Dead Code:** this code smell can be found in Spool class (line 22). We can find in this class that portions of this class are never being executed. The only solution that can be applied is to eliminate (delete) this part of code that is never used or executed.
- 4) **Dead Code:** This code smell can be found in Printer class (line 20). In this particular case like as the case exposed before we can find portions of code that are not being used or executed. The only solution that can be applied could be to eliminate (delete) this part of code that is never executed or used.

## Week 2: Refactoring

### 2.1 Applying software principles

- *Apply the single-responsibility principle, open-closed and interface segregation to the code of the 3D printer casus. Refactor the code to a better solution. In your report write how you applied each principle: prove to the reader that you made the code better!: show code fragments, discuss your solution, show class diagrams etc.*

First of all, we will start describing with our own words what are these SOLID principles mentioned previously.

- **Single Responsibility Principle (SRP):** This principle says that a class should have only one reason to change, that is, it should have a single responsibility. Each class must be in charge of a single task or concept within the system.
- **Open-Closed Principle (OCP):** The open-closed principle says that a class should be open for extension but closed for modification. This means that new functionality can be added to a system without changing existing code. This also means that code can be extended to fit new needs without altering their original source code, this reduces the risk of introducing bugs into the existing code of the system.
- **Interface Segregation Principle (ISP):** The interface segregation principle holds that a class should not be forced to implement interfaces that it doesn't use. In other words, a class should not be forced to depend on interfaces that contain methods it doesn't need. Instead of having large, monolithic interfaces, it is preferable to have smaller, specific interfaces.

After describing these three SOLID principles, now we will apply them in our exam case.

This principle states that a class should have only one reason to change, meaning that a class should only have one job or responsibility.

Let's start by refactoring the PrinterManager class. It currently has responsibilities related to managing printers, prints, and print tasks. We'll create separate classes for these responsibilities.

New Classes:

- PrintQueueManager: Manages the print queue, selection of print tasks, and starting the initial queue.

```
10 public class PrintQueueManager {
    9 usages
11     private final List<Printer> printers;
    4 usages
12     private final List<Spool> spools;
    1 usage
13     private List<Spool> freeSpools = new ArrayList<>();
    5 usages
14     private final List<Print> prints;
    7 usages
15     private final PrintTaskManager printTaskManager;
    3 usages
16     private List<Printer> freePrinters = new ArrayList<>();
    5 usages
17     private final PrinterEventManager printerEventManager;
    2 usages
18     private List<PrintTask> pendingPrintTasks;
    2 usages
19     private String printStrategy = "Less Spool Changes";
    2 usages  ⚡ RubenGomezHidalgo
20     public PrintQueueManager() {
21         this.printers = new ArrayList<>();
22         this.spools = new ArrayList<>();
23         this.prints = new ArrayList<>();
24         this.printTaskManager = new PrintTaskManager();
25         this.printerEventManager = new PrinterEventManager();
26         this.pendingPrintTasks = new ArrayList<>();
    }
```

- PrintTaskManager: Manages print tasks, including adding, removing, and selecting tasks.

```
2 usages  ⚡ RubenGomezHidalgo
9 public class PrintTaskManager {
    4 usages
10     private final List<PrintTask> printTasks;
11
    1 usage  ⚡ RubenGomezHidalgo
12     public PrintTaskManager() {
13         this.printTasks = new ArrayList<>();
14     }
15 }
```

- PrinterEventManager: Handles events like printer failure and completion.

```
2 usages  ▴ RubenGomezHidalgo
8 public class PrinterEventManager {
    2 usages
9     private List<PrintTask> activePrintTasks;
    4 usages
10    private Map<Integer, Integer> failedPrinters;
11
12    1 usage  ▴ RubenGomezHidalgo
13    public PrinterEventManager() {
14        this.activePrintTasks = new ArrayList<>();
15        this.failedPrinters = new HashMap<>();
16    }
```

## 2.2 Facade pattern

- **Are there problems in the code that could be resolved by using the facade pattern?**
  1. One of the things that could benefit us when applying a facade pattern is to be able to simplify the user's interaction with this system
  2. The current code requires the client to interact directly with different classes such as `PrinterQueueManager`. Implementing a facade would hide the internal complexity of these classes and provide only the essential operations to the client. The `PrintQueueManager` delegates the task of adding a print task to the queue to the selected strategy. This way, you can dynamically change the behavior of your print queue management without altering the client code or the `PrintQueueManager` class.
- **What are these problems, and where in the code can you find these problems?**

The current problem is that the user has a direct interaction with the `PrinterQueueManager` class through the MAIN class. The facade pattern would decouple this direct interaction with the system, thus allowing a better concealment of the complexity of the entire system.
- **Does the facade pattern solve these problems? if so, how?**

Yes, since it allows us to hide the complexity from the user who interacts with the system.
- **Implement the facade pattern in the code**

To apply the Facade pattern in the Main, we could create a new class called `PrintManagerFacade` that encapsulates and simplifies operations related to

printers, prints, and print tasks. This class will act as a facade for the printing system.

Then, in the Main class, we will instantiate and use the new PrintManagerFacade class, instead of interacting directly with PrintQueueManager.

This refactoring will simplify our Main class and improve code organization by separating printing system operations into a dedicated facade.

### PrintManagerFacade Class

```
11      2 usages  ⬆ RubenGomezHidalgo
      public class PrintManagerFacade {
12          2 usages
      private final Scanner scanner;|
13          19 usages
      private PrintQueueManager printQueueManager;
14
15          1 usage  ⬆ RubenGomezHidalgo
      public PrintManagerFacade() {
16          this.scanner = new Scanner(System.in);
17          this.printQueueManager = new PrintQueueManager();
18      }
```

### Main Class

```
32
33      int choice;
34      do {
35          showMenu();
36          choice = menuChoice( max: 9);
37
38          switch (choice) {
39              case 1:
40                  printManagerFacade.addNewPrintTask();
41                  break;
42              case 2:
43                  printManagerFacade.registerPrintCompletion();
44                  break;
45              case 3:
46                  printManagerFacade.registerPrinterFailure();
47                  break;
48              case 4:
49                  //Strategy Pattern
50                  printManagerFacade.changePrintStrategy();
51                  break;
52              case 5:
53                  printManagerFacade.startPrintQueue();
54                  break;
55              case 6:
56                  printManagerFacade.showPrints();
```



- **Convince the reader that you made the code better**

The Facade Pattern is beneficial for the code because it simplifies the interaction between various components and provides a unified interface to the user, making the system easier to use and understand. In the context of the provided code, the Facade Pattern is applied to create a `PrintManagerFacade`, serving as a simplified entry point for the user to interact with the print management system.

The `PrintManagerFacade` acts as a simplified interface for the user. Instead of dealing with multiple classes and their complexities, users interact with a single, well defined facade class. This simplification enhances the usability of the system.

Also, Facade Pattern will encapsulate the complexity. Users don't need to be concerned with the intricacies of individual classes and their interactions. This encapsulation promotes a cleaner and more understandable codebase.

Readability will improve by providing a clear and concise set of methods in the facade. Users can easily identify the available functionalities and understand how to interact with the print management system.

Finally, users are less dependent on the internal structure of the system. This reduces some errors caused by incorrect usage of individual components. It also facilitates changes to the internal structure without affecting users.

## Week 3: Design patterns

### 3.1 Strategy and singleton pattern

#### 3.1.1 Strategy pattern

- **Are there problems in the code that could be resolved by using the strategy pattern?**

The Strategy Pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. It allows the client to choose the appropriate algorithm at runtime.

In our project, we have a "Change printing style" menu option. If this involves changing the way prints are handled or queued, you can use the Strategy Pattern to encapsulate different print queueing strategies.

- ***What are these problems, and wherein the code can you find these problems?***

This problem can be found in our Main Class in specifically in the "Change printing style".

- ***Does the strategy pattern solve these problems? if so, how?***

Yes, The PrintQueueManager delegates the task of adding a print task to the queue to the selected strategy.

This way, you can dynamically change the behavior of your print queue management without altering the PrintQueueManager class.

- ***Implement the strategy pattern in the code***

The PrintQueueManager delegates the task of adding a print task to the queue to the selected strategy.

This way, you can dynamically change the behavior of your print queue management without altering the client code or the PrintQueueManager class.

#### PrintStrategy Interface

```
7 usages 2 implementations new *
6 public interface PrintStrategy {
  1 usage 2 implementations new *
7 void addToQueue(PrintQueueManager printQueueManager, PrintTask printTask);
8 }
9
```

## EfficientSpoolChangesStrategy Class

```
6 public class EfficientSpoolUsageStrategy implements PrintStrategy {
7
8     1 usage new *
9     @Override
10     public void addToQueue(PrintQueueManager printQueueManager, PrintTask printTask) {
11         // Implement strategy specific logic
12         // ...
13         printQueueManager.addPrintTaskToQueue(printTask);
14     }
15 }
```

## LessSpoolChangesStrategy Class

```
5 public class LessSpoolChangesStrategy implements PrintStrategy {
6     1 usage new *
7     @Override
8     public void addToQueue(PrintQueueManager printQueueManager, PrintTask printTask) {
9         printQueueManager.addPrintTaskToQueue(printTask);
10     }
11 }
```

## Main Class

```
109 // Start Strategy Pattern
110 no usages 1 carlosguinovart+1
111 private void changePrintStrategy() {
112     System.out.println("----- Change Strategy -----");
113     System.out.println("- Current strategy: " + printQueueManager.getPrintStrategy());
114     System.out.println("- 1: Less Spool Changes");
115     System.out.println("- 2: Efficient Spool Usage");
116     System.out.print("- Choose strategy: ");
117     int strategyChoice = numberInput(1, 2);
118
119     if (strategyChoice == 1) {
120         printQueueManager.setPrintStrategy("Less Spool Changes");
121     } else if (strategyChoice == 2) {
122         printQueueManager.setPrintStrategy("Efficient Spool Usage");
123     }
124
125     System.out.println("-----");
126 }
127 // Finish Strategy Pattern
```

```
48 case 4:
49     //Strategy Pattern
50     printManagerFacade.changePrintStrategy();
51     break;
```

- ***Convince the reader that you made the code better***

The Strategy Pattern is a beneficial design pattern for the code because it promotes flexibility, maintainability, and extensibility. In the context of the provided code, the Strategy Pattern is applied to the print queue management system to allow for dynamic selection of different printing strategies without altering the existing codebase.

With the Strategy Pattern, we will have flexibility because it can encapsulate different print queue management strategies into separate classes. This allows us to easily add new strategies or modify existing ones without impacting the rest of the code.

Improves code maintainability by organizing the code into distinct strategies. Each strategy is responsible for a specific aspect of print queue management, making it easier to understand, modify, and extend the system.

New printing strategies can be added without modifying existing code. This is particularly useful when introducing new requirements or optimizations to the print queue management system.

The use of concrete strategy classes with clear names (LessSpoolChangesStrategy, EfficientSpoolUsageStrategy) enhances code readability. It makes it easier for developers to comprehend the purpose of each strategy and how it contributes to the overall functionality.

Also, it is a code readability because of the classes. On the other hand, it allows us to switch between different strategies at runtime. This adaptability is beneficial when the system needs to respond dynamically to changing requirements or user preferences.

### 3.1.2 Singleton pattern

- **Are there problems in the code that could be resolved by using the singleton pattern?**

Based on the code snippets and information provided, there is no explicit indication that the Singleton Pattern is needed for any specific class in the project. The Singleton Pattern is typically used when there should be exactly one instance of a class, providing a global point of access to that instance.

- ***What are these problems, and wherein the code can you find these problems?***

The codebase doesn't exhibit issues that the Singleton pattern would typically address.

The Singleton pattern is primarily used when you want to ensure a class has only one instance and provides a global point of access to that instance.

- ***Does the singleton pattern solve these problems? if so, how?***

In this particular project, the absence of the Singleton pattern is not a problem. The Singleton pattern is useful when there should be only one instance of a class, and that instance needs to be accessible globally. If the project doesn't require such a constraint, applying the Singleton pattern might introduce unnecessary complexity.

- ***Implement the singleton pattern in the code***

Since there isn't a specific problem in the code that necessitates the Singleton pattern, implementing it would be inappropriate and could potentially complicate the design without providing tangible benefits.

- ***Convince the reader that you made the code better***

The code has been improved by introducing design patterns such as the Adapter pattern. This choice was made purposefully to address specific challenges, like seamlessly integrating new functionalities and ensuring a consistent interface. The Adapter pattern enhances modularity, readability, and extensibility in a way that aligns with the project's requirements. While the Singleton pattern wasn't deemed necessary for this context, leveraging appropriate patterns like Adapter contributes to a more robust and maintainable codebase.

## 3.2 Adaptor and factory (TODO: Richard)

### 3.2.1 Adaptor pattern

- **Are there problems in the code that could be resolved by using the Adaptor pattern?**

Certainly, applying the Adapter Pattern involves creating adapters to make different interfaces compatible. However, to implement this pattern effectively, we need to identify specific scenarios where different interfaces need to be adapted. Let's take a closer look at the provided code and identify potential areas where the Adapter Pattern might be beneficial.

- ***What are these problems, and wherein the code can you find these problems?***

The Adapter pattern can be used to make existing classes compatible with the Printable interface.

Adapters act as intermediaries, allowing existing classes to work with the new interface without modifying their code.

- **Does the Adaptor pattern solve these problems? if so, how?**

Yes, the Adapter pattern helps solve the identified problems in the code.

First issue: The Printable interface is introduced later, and existing classes like Printer may not implement it.

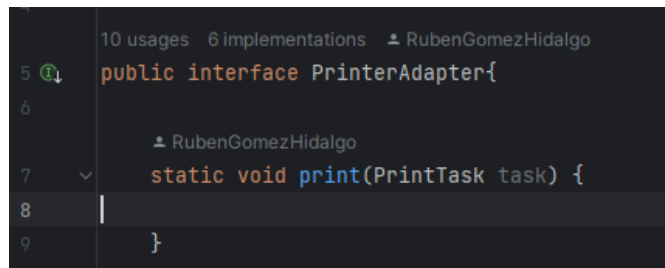
Adapter Solution: The PrinterAdapter acts as an adapter, implementing the Printable interface and delegating calls to the existing Printer class. This allows the Printer class to work seamlessly with the Printable interface.

Integration with New Implementations:

Second Issue: Adding new types of printers or printing devices may not be straightforward without modifying the existing code.

Adapter Solution: The PrinterAdapter encapsulates the logic needed to adapt the existing Printer class to the new Printable interface. When introducing new printer implementations, you can create adapters for them, ensuring they also conform to the Printable interface. This promotes a modular and extensible design, allowing easy integration of new printers without modifying existing code.

- **Implement the Adaptor pattern into the code**

A screenshot of an IDE showing the definition of the `PrinterAdapter` interface. The interface is located in a file named `PrinterAdapter.java` by `RubenGomezHidalgo`. It has 10 usages and 6 implementations. The code is as follows:

```
5 public interface PrinterAdapter{  
6  
7     static void print(PrintTask task) {  
8  
9     }  
}
```

- **Convince the reader that you made the code better**

Before the Adapter pattern, integrating the existing `Printer` class with the new `Printable` interface could have been a challenge. However, the Adapter pattern acts as a bridge, allowing the `Printer` class to seamlessly work with the `Printable` interface. This ensures smooth integration between old and new code components.

Introducing new printer implementations is now a breeze. Instead of modifying the existing codebase, we create specific adapters like `PrinterAdapterImpl`. These adapters make new implementations adhere to the `Printable` interface, promoting a modular and extensible design.

Adapters encapsulate adaptation logic, making it reusable for different implementations. This not only reduces redundancy but also ensures that the adaptation process is consistent across various scenarios, contributing to a more maintainable codebase.

Ensuring a consistent interface across different printer implementations is crucial. The `Printable` interface provides a unified contract, and adapters ensure that both existing and new implementations adhere to this common interface. This consistency simplifies interactions and usage throughout the codebase.

### 3.2.2 factory pattern

- *Are there problems in the code that could be resolved by using the factory pattern?*

Yes, there are some parts in the code that could be improved by using this pattern.

Right now, creating objects is kind of scattered around the code. This can make things messy and hard to manage. With the Factory pattern, we put all the object creation stuff in one place, making it easier to understand and maintain.

- *What are these problems, and wherein the code can you find these problems?*

Printer objects are created in the Main class when reading data from files. This logic could be centralized in a single location, making it easier to modify and maintain.

We have the same situation with printers, the creation of print tasks is also scattered across different methods in the Main class. Consolidating this logic into a factory would improve code organization and readability.

Spool objects are also created in the Main class, adding to the scattered object creation problem.

- *Does the factory pattern solve these problems? if so, how?*

We could apply the Factory pattern to create different types of objects such as printers, prints, or spools. For example, we could have a PrintFactory responsible for creating different types of prints based on certain criteria, or a PrinterFactory for creating different types of printers.

- *Implement the factory pattern into the code.*

```
package nl.saxion.Factory;

import nl.saxion.Models.MultiColor;
import nl.saxion.Models.Printer;
import nl.saxion.interfaces.PrinterFactory;

// usages new *
public class MultiColorPrinterFactory implements PrinterFactory {
    // usages new *
    @Override
    public Printer createPrinter(int id, String name, String manufacturer, int maxX, int maxY, int maxZ, int maxColors) {
        return new MultiColor(id, name, manufacturer, maxX, maxY, maxZ, maxColors);
    }
}
```



```

package nl.saxion.Factory;

import nl.saxion.Models.Printer;
import nl.saxion.Models.StandardFDM;
import nl.saxion.interfaces.PrinterFactory;

no usages new *
public class StandardPrinterFactory implements PrinterFactory {
    1 usage new *
    @Override
    public Printer createPrinter(int id, String name, String manufacturer, int maxX, int maxY, int maxZ, int maxColors) {
        new *
        return new StandardFDM(id, name, manufacturer, maxX, maxY, maxZ) {
        };
    }
}

```

```

package nl.saxion.interfaces;

import nl.saxion.Models.Printer;

6 usages 2 implementations new *
public interface PrinterFactory {
    1 usage 2 implementations new *
    Printer createPrinter(int id, String name, String manufacturer, int maxX, int maxY, int maxZ, int maxColors);
}

```

- *Convince the reader that you made the code better.*

The factory pattern encapsulates the creation logic of objects within dedicated factory classes. This means that the “Main” class no longer needs to know the intricate details of how each type of printer is created. This encapsulation leads to cleaner, more modular code.

Adding new types of printers in the future becomes much easier with the factory pattern. By creating new factory implementations for each type, we can integrate additional printer types into the system without affecting existing code. This promotes scalability, allowing the codebase to evolve with changing requirements.

## Week 4: More design patterns

### 4.1 Observer pattern

- *Write about the Observer pattern:*

It's a software design pattern that defines 1:N(one-to-any) dependency between objects, that when an object changes state, all of the dependents are automatically notified and updated. This pattern belongs to the category of behavioral patterns, as it deals with the distribution of responsibility between an object and its observers.

Components of this pattern:

- **Subject:** is the observed object. This object has a list of observers and notifies these of any changes in their status.
  - **Observer:** it's the interface that defines the update method that observers must implement to receive notifications from the Subject.
  - **Concrete observers:** are classes that implement the observer interface. These classes want to be notified when there are some changes in the subject.
- *What problems did this implementation solve? Why is the observer a good idea here?*

By implementing the observer pattern, the 3D print scheduler can notify other components (observers) of changes in the print queue without requiring tight coupling between them.

Also, the observer pattern allows new observers to be added to the system easily, enabling the integration of additional functionalities or monitoring capabilities without modifying existing code.

By separating concerns and responsibilities through the observer pattern, the codebase becomes more modular and easier to understand.



## Week 5: Expansion of the system

5.1 Testing

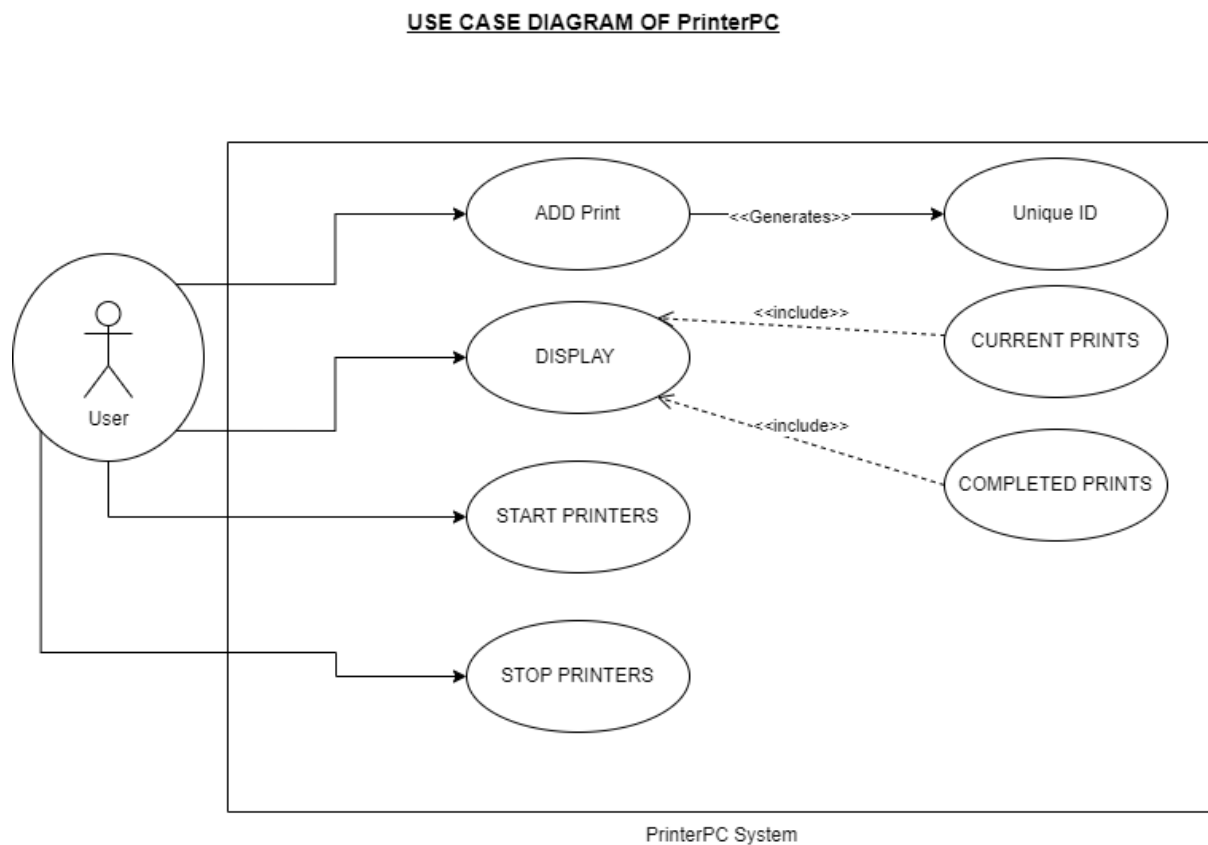
5.2 Conclusion

# Part 2: Software architecture

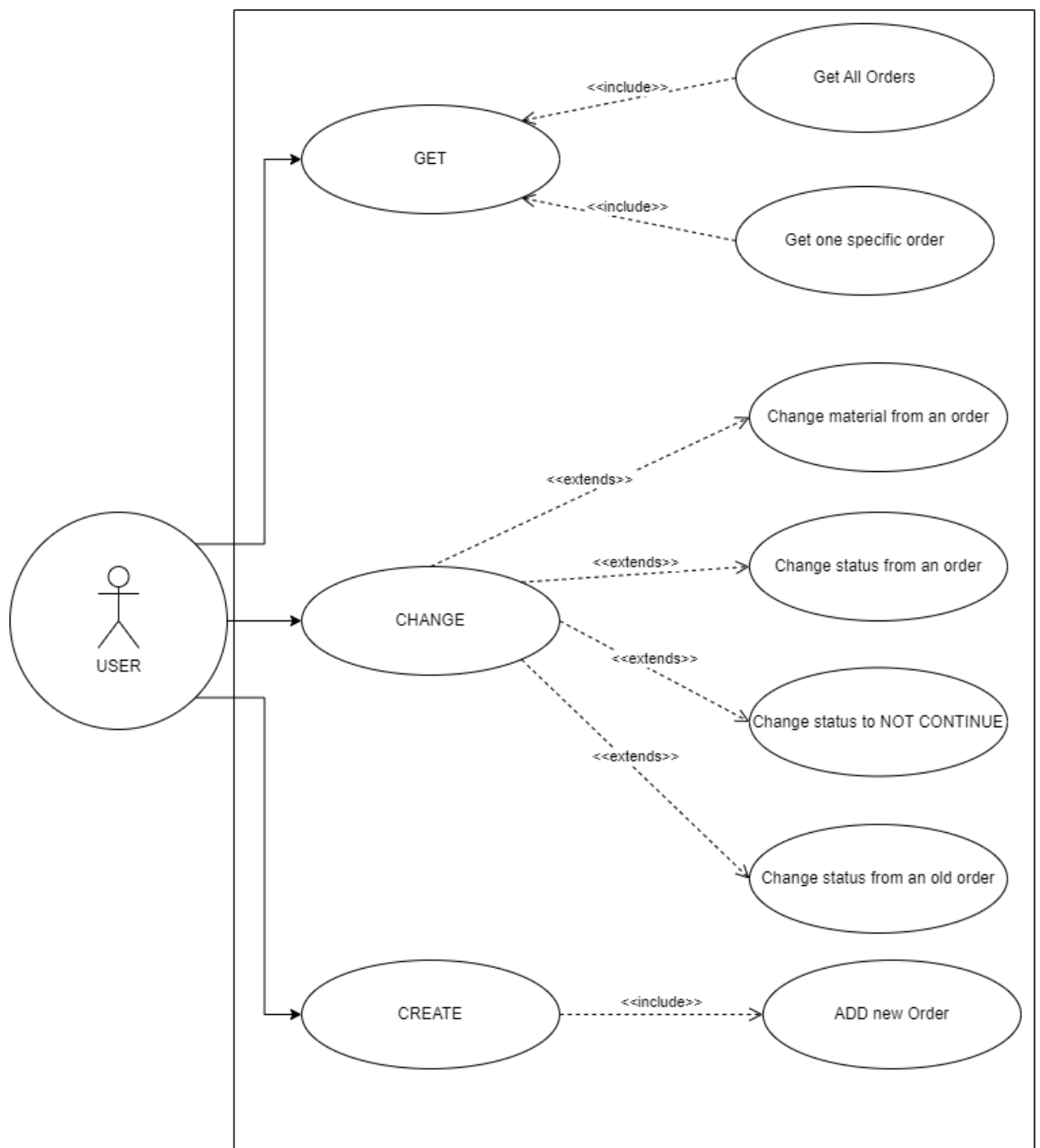
## Week 6: Architecture

### 6.1 Architecture

#### Use case diagram of PrinterPC



## Use case diagram of Webshop



WebShop System Use Case

- *Create a sketch of the system*

#### **PrinterPC Interface system sketch:**

Split problems into a small part:

- Add print to be printed → returns a unique ID
- Start printing process.
- Stop printing process.
- Display current prints.
- Display completed prints.

Key physical components:

- User
- PrinterPC program
- Printers (up 24)

Who needs to interact with the system?

- User (employees)

How are they going to interact with the system?

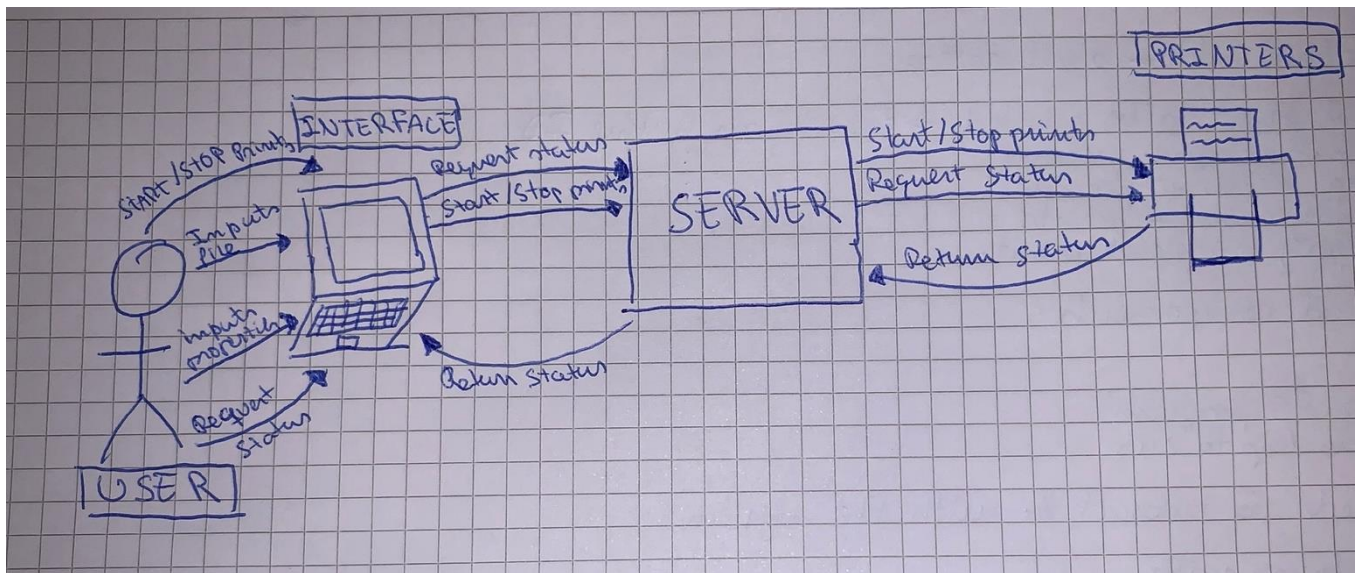
- PC with the program

How are the components connected?

- We have thought of putting a server between the pc and the printers where all the printer status data will be stored.

Think about who has initiative and which part is responsible for which data/behavior:

- USER>PROGRAM → will enter data (file, properties) to the program needed to start/stop prints and view status of current/completed prints.
- PROGRAM>SERVER → will send data to server to process all the requests that user enter to the program.
- SERVER>PRINTER → will process the data received and send the necessary commands to the printers as user wanted to.



### Webshop system sketch:

Split problems into a small part:

- Get all orders.
- Get one specific order.
- Change material from an order.
- Change status from an order.
- Change status to NOT continue.
- Change status from an old order.
- Get the last order created.

Key physical components:

- User
- Computer/phone
- Printer
- Server

Who needs to interact with the system?

- User

How are they going to interact with the system?

- Web portal via a browser

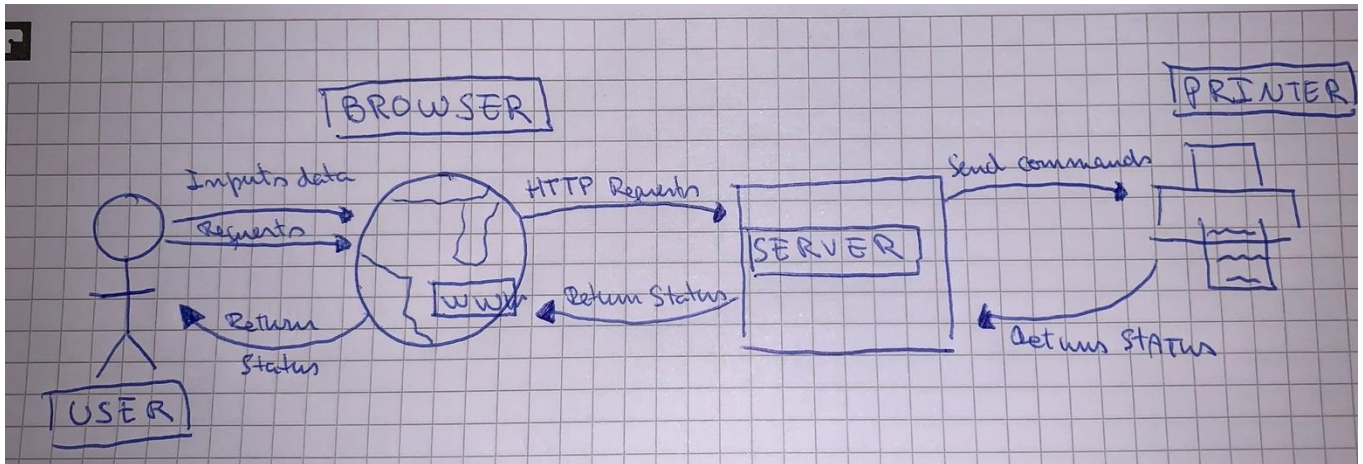
How are the components connected?

- The user interacts with a web portal from a browser, this web portal is directly connected to a server that will process the user's requests. after processing the requests, it will send the necessary commands to the printers.

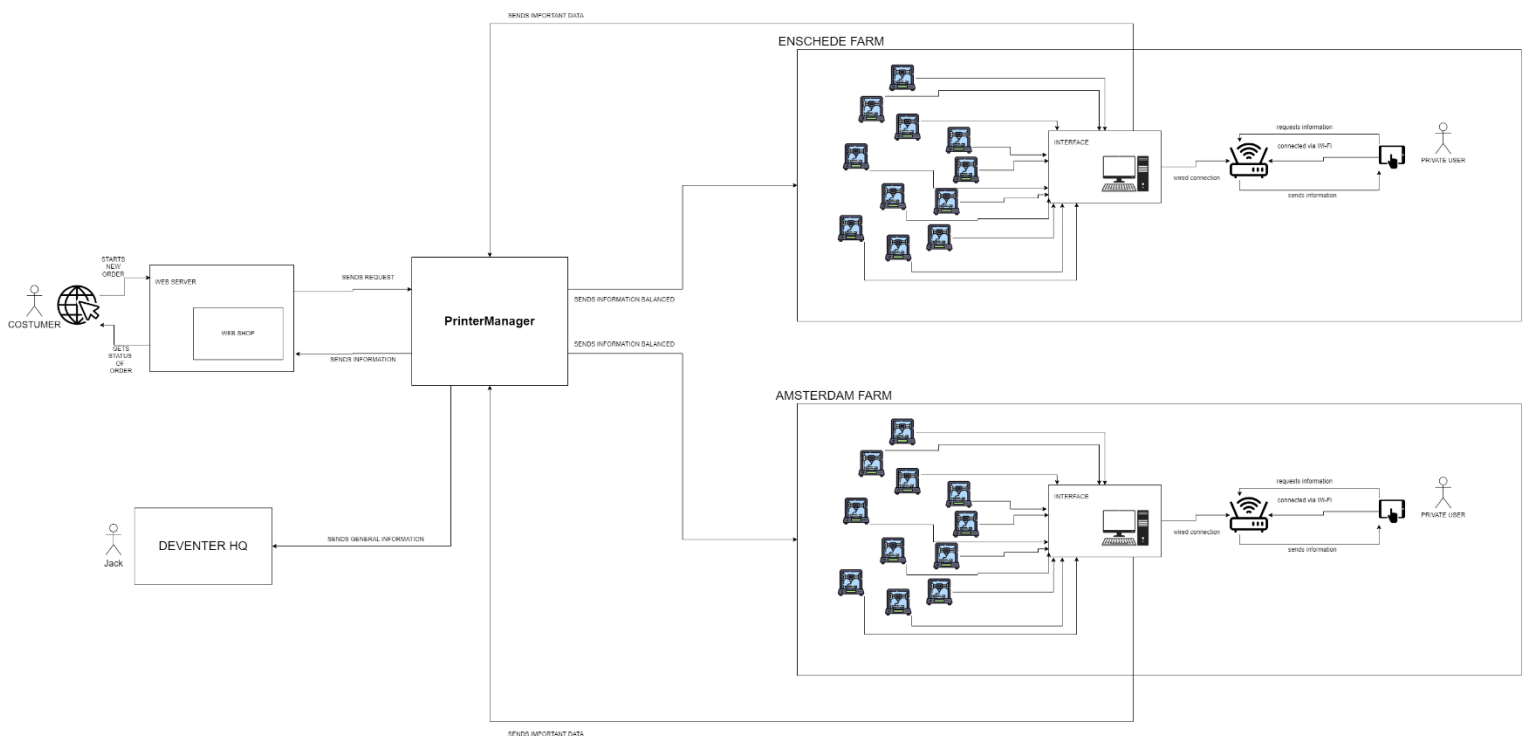
Think about who has initiative and which part is responsible for which data/behavior:



- USER>BROWSER → will enter data (file, properties) to the program needed to start/stop prints and view status of current/completed prints.
- BROWSER>SERVER → will send data to server to process all the requests that user enter to the program.
- SERVER>PRINTER → will process the data received and send the necessary commands to the printers as user wanted to.



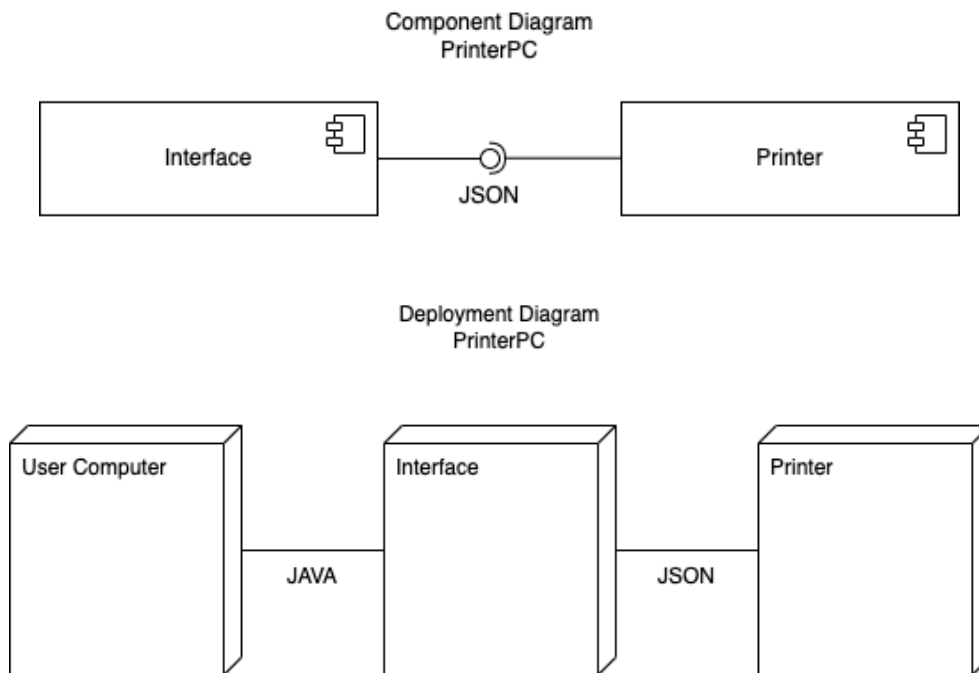
## Global system sketch:



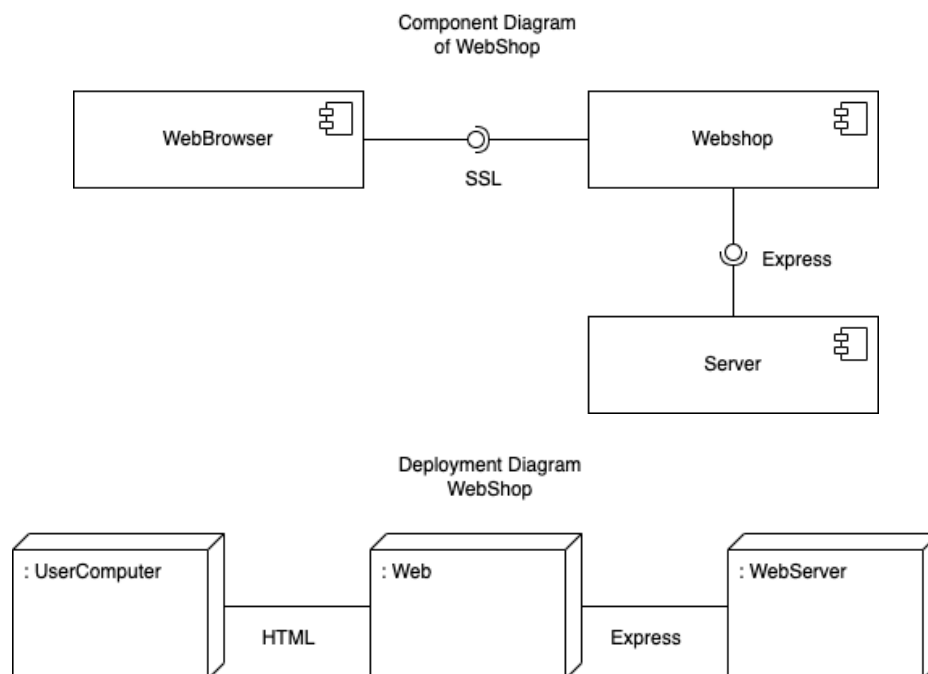
In this general sketch that we have created, what we want to highlight are the following connections:

- The **printManager** would be like the program that manages the whole issue of impressions. This one also has an algorithm that decides by itself which farm each impression that comes to it by order from the WebShop should go.
- We have implemented that **Jack** from DEVENTER can directly consult the states of the impressions with an overview.
- We have added a communication between this general manager and the "interfaces" that manage the printing farms.
- **Management interfaces:** we decided that there would be 2 farm management interfaces (one for each farm). These give direct information to the employees who are responsible for controlling the correct operation in each farm.
- For this management of employees (private users), as required, we have designed the system to communicate with employees using **tablets**. That's why we made sure that from the computer that manages the interfaces, there is a **wireless connection** with the **tablets**. Hence the two icons of wireless access points.
- The information that each customer will see is the status of their order.

## Component and deployment diagram PrinterPC interface



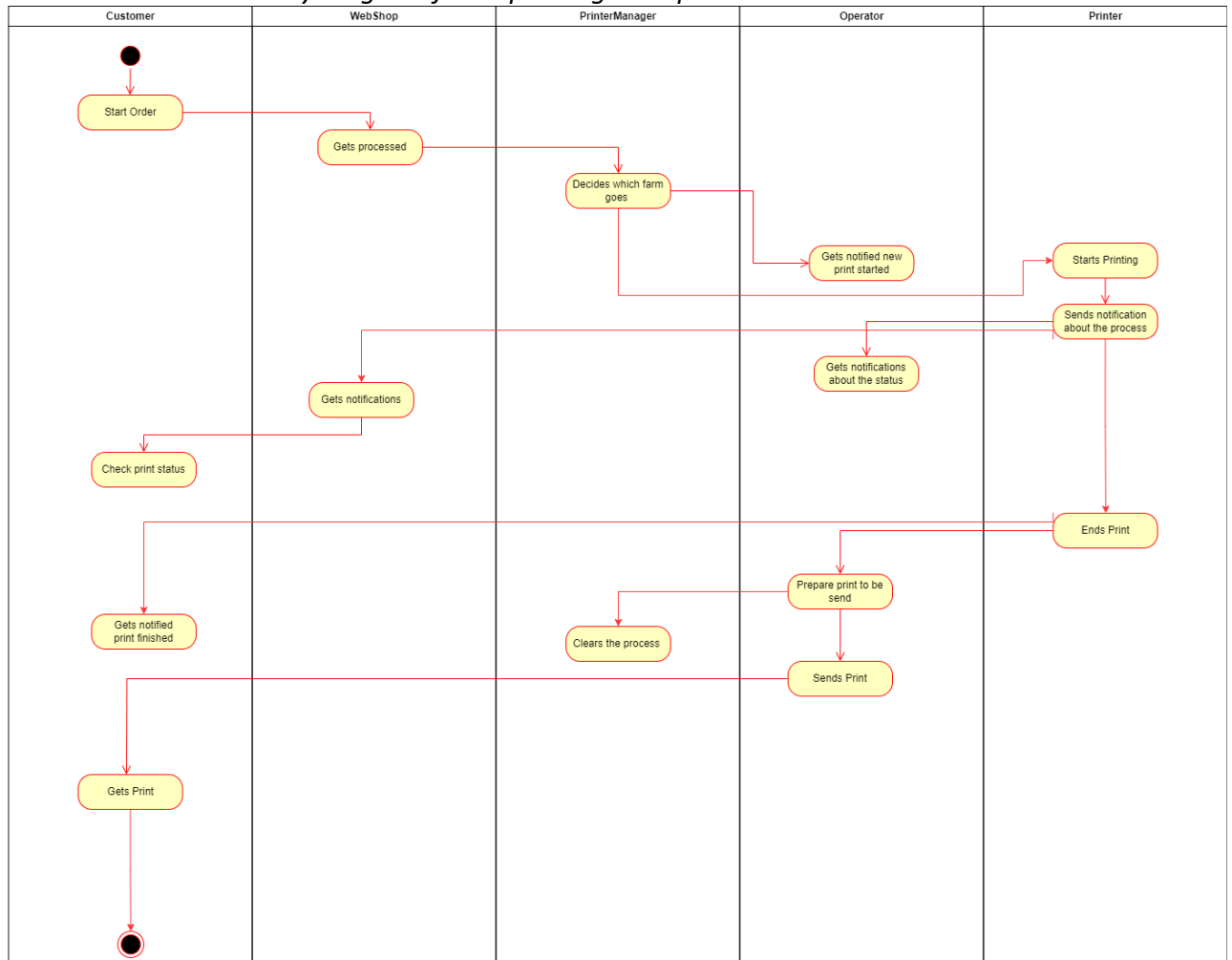
## Component and deployment diagram Webshop interface



## 6.2 Connection between components and activity diagrams

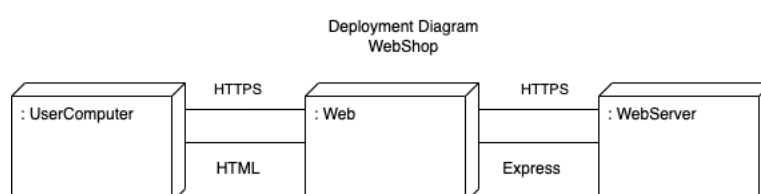
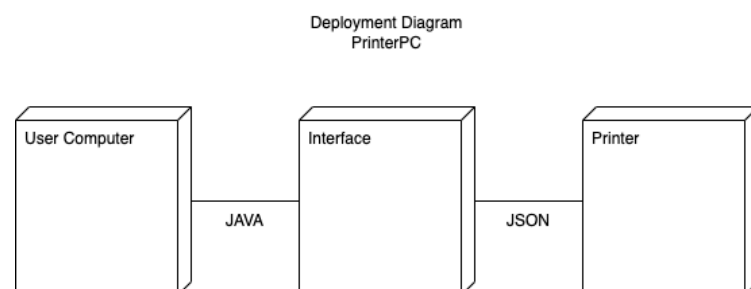
### 6.2.1 activity diagram

- *Create an activity diagram for requesting a 3D print.*



### 6.2.2 architectural patterns

- *Add connection types to your component/deployment diagram.*



- *Consider adding the architectural patterns discussed in week 6.2*

We have decided that in both systems there is a pattern of Message Queues, as we ensure that a single print goes to a single printer. If we chose Publisher/subscriber, what I would do would be that all printers would receive the command to print the same project. Since the publisher would be the program (interface/webshop) and the subscribers would be the printers.

We have decided to implement Server-Side Load Balancing. This means that we will place a dedicated load balancing software between the clients and the servers. The load balancer receives incoming requests and forwards them to the appropriate farm based on the algorithm. This allows for centralized control and management of traffic distribution.

This algorithm will work based on client proximity and print requests will be sent to the nearest printer farm to their location. For example, if the client lives near Enschede, then everything will automatically be sent to Enschede so they can pick it up there if they require it. This will provide convenience for all clients.

What this algorithm also takes into account, is the option of how they want to receive the print, if by postal service or if they want to come and collect it themselves at our farms. What the algorithm will do is that if it is to be sent by postal service, it will choose where to print depending on the load of each farm. For example Tonny who lives in Groningen and has chosen that he wants to receive the figurine by postal service, the algorithm will look at which of the 2 farms has the least workload.

We had considered implementing a round robin in queue management. This possibility has been abandoned since we have realized that we did not take into consideration the customers who wanted to come and collect their impressions directly at the farm.

## Week 7 + 8 Proof of concept

In the printerPc-main project, there are different parts that help manage a printer system. It's divided into two main sections: Models and Data.

In the Models section, there are classes that describe print jobs. One class, called Print, stores details like the name of the print job, how long it takes, and how many colors it needs. Another class, ActivePrint, keeps track of print jobs that are currently being worked on. It remembers things like the job's name and how much time is left.

In the Data section, there are classes that handle loading print job information. For instance, there's a class called JSONPrintLoader that reads print job details from a special file called "prints.json". This class helps the program understand what print jobs are available.

The main part of the program is the PrinterPCMock class. It pretends to be a printer system by handling things like adding new print jobs, checking which print jobs are active, and completing print jobs. It also has a way to start and stop the pretend printer.

The project is designed to be flexible, meaning it can handle different situations. For example, it can easily switch between different ways of loading print job information. There are also parts of the code specifically meant for testing, to make sure everything works as expected.

Overall, the printerPc-main project is set up to help manage a printer system in a simple and organized way, making it easier to understand and use.