SEC-TMP-DLD

**S/W Detailed Level Design**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Name** |  | | |
| **Block Name** |  | | |
| **Author** |  | **Approver** |  |
| **Team** |  | | |

This document represents Detailed Level Design (DLD). It describes the detailed system design and implementation plan in alignment with Agile principles. The DLD is updated incrementally with each release to reflect system evolution.

Contents

[1. Overview 4](#_Toc210821152)

[2. System Overview / Architectural Context 5](#_Toc210821153)

[3. UML Class Diagram (Technical Design) 6](#_Toc210821154)

[4. Class Specifications 6](#_Toc210821155)

[5. Interfaces and Abstractions 6](#_Toc210821156)

[6. Function Responsibilities 6](#_Toc210821157)

[7. Operation Flow 7](#_Toc210821158)

[8. Enumerations & Constants 7](#_Toc210821159)

[9. Validation Rules & Future Work 7](#_Toc210821160)

[10. Traceability Matrix 7](#_Toc210821161)

[11. Code Structure and File Mapping 7](#_Toc210821162)

[12. Revision History 8](#_Toc210821163)

* Revision History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Version | Date | Revised contents | Author | Approver |
|  |  |  |  |  |
|  |  |  |  |  |

* Terms and Abbreviations

|  |  |
| --- | --- |
| **Term** | **Description** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

* References

1. SW Requirements Specification

# 

# Overview

<This section provides an overview of the system and its purpose. It defines the scope, main objectives, and the role of this document within the project. It should briefly describe the system context and identify the key stakeholders and user roles.>

The main purpose of this software is to optimize the workflow of the Photo Studio project.

The software system will also record orders in a centralized manner, track their completion and the related income and expenses for logging and to give a revenue report to the studio administrator.

The system is meant to allow the receptionist to:

* Create clients in the system
* Easily create orders with the client data.
* Generate automatically two types of orders (Client and photographer)
* Easy access to client orders.
* Automatically calculate the price (meaning the surcharge) and mark an express order automatically.
* Have centralized mappings with all the orders recorded.
* Create and submit the daily revenue reports easily.

The system will allow the photographer to:

* Submit a report on consumed materials at the end of the day.
* Submit the completion of an order.

The intended users are:

* the client
* the receptionist
* the photographer
* the studio administrator

## Scope:

The system includes:

* Creation of clients.
* Creation of the orders.
* The transmission of orders (receptionist -> Photographer, receptionist -> Client).
* Price calculation based on order deadline (Mark it as express if necessary).
* Track of the consumables materials.
* Submissions of orders completed by the photographer.
* Generation of the reports (Revenue and consumables) with getters, so the administrator can view them whenever they want.

The system does not include:

* The development of film/printing images.
* Payment system.
* The transfer of printed images/developed film from the photo studio to the customer.
* Replenishing the materials

Benefits:

* Quick distribution of orders
* Easy reporting/concise overview for administrator
* Automate the generation of the reports

Key features:

* Automatic price increase on orders in case of urgent orders
* Generation of different orders for the photographer and the client
* Generation of the reports

# System Overview / Architectural Context

<High-level description of the architecture and components. A simplified diagram can be included to show layers (e.g., UI, Service, Data) or main modules. This section establishes the overall design philosophy and dependencies.

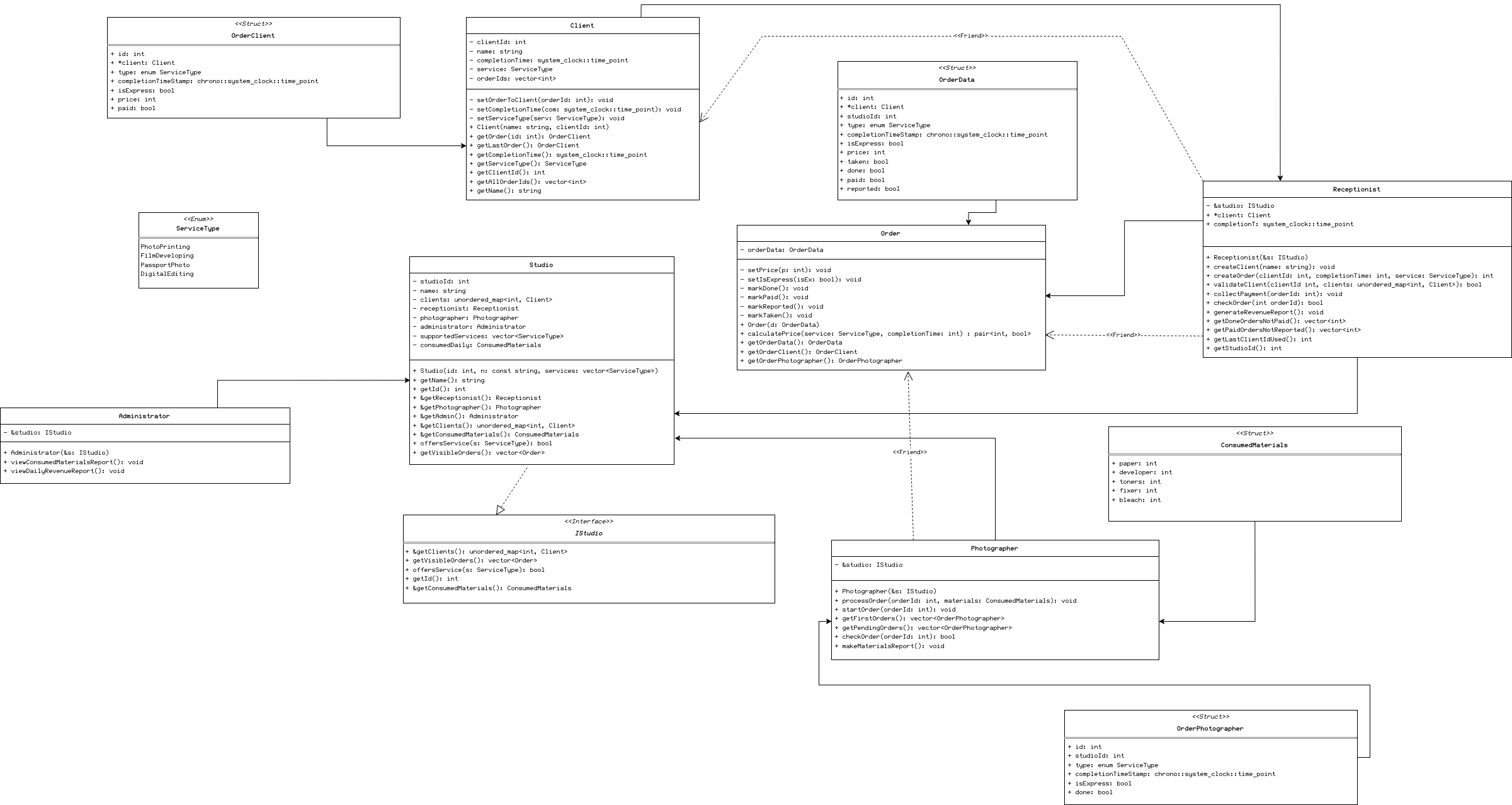
Example:  
- Presentation Layer (UI / ConsoleUI)  
- Logic Layer (Manager, Services)  
- Data Layer (Repositories, FileService)  
Each layer interacts only with the one directly below it. Include a simple schematic or diagram to illustrate dependencies (UI → Logic → Data).>

Presentation layer – CLI

Logic layer – Administrator, Photographer, Receptionist, Report generation

Data Layer – text files for storing and retrieving reports

# UML Class Diagram (Technical Design)



# Class Specifications

<Description:  
Detailed description of each class. For every class, specify:

* Purpose (responsibility)
* Attributes (with types and short explanation)
* Methods (signatures and short description)
* Constraints / Contracts (preconditions, postconditions, invariants if applicable)

A tabular format is recommended for readability, for example: >

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Type | Description | Attributes | Methods |
| Client | Client | Store information about client’s order(s) | clientId: int  name: string  completionTime: system\_clock::time\_point  service: ServiceType  orderIds: vector<int> | Client(string name, int clientId)  SetOrderToClient(int orderId)  setCompletionTime(system\_clock::time\_point com)  setServiceType(ServiceType serv)  getOrder(int id) → OrderClient  getLastOrder() → OrderClient  getCompletionTime() → system\_clock::time\_point  getServiceType() → ServiceType  getClientId() → int  getAllOrderIds() → vector<int>  getName() → string |
| Receptionist | Studio Employee | Creates clients and orders, reports on daily revenue | &studio: Istudio  \*client: Client  completionT:  system\_clock::time\_point | Receptionist(&s: Istudio)  createClient(name: string)  createOrder(int clientId, int completionTime, ServiceType service) → int  validateClient(int clientId, unordered\_map<int, Client> clients) → bool  collectPayment(int orderId)  checkOrder(int orderId) → bool  generateRevenueReport()  getDoneOrdersNotPaid() → vector<int>  getPaidOrdersNotReported() → vector<int>  getLastClientIdUsed() → int  getStudioId() → int |
| Order | Order | Calculates price (express or not), stores order data | orderData: OrderData | Order(OrderData d)  setPrice(int p)  setIsExpress(bool isEx)  markDone()  markPaid()  markReported()  markTaken()  calculatePrice(ServiceType service, int completionTime) → pair<int, bool>  getOrderData() → OrderData  getOrderClient() → OrderClient  getOrderPhotographer() → OrderPhotographer |
| Photographer | Studio employee | Start and process orders, validate orders to be completed | &studio: IStudio | Photographer(Istudio &s)  processOrder(int orderId, ConsumedMaterials materials)  startOrder(int orderId)  getFirstOrders() → vector<OrderPhotographer>  getPendingOrders() → vector<OrderPhotographer>  checkOrder(int orderId) → bool  makeMaterialsReport() |
| Administrator | Studio employee | View report of consumed materials and daily studio revenue | &studio: IStudio | Administrator(Istudio &s)  viewConsumedMaterialsReport()  viewDailyRevenueReport() |
| Studio | Studio | Contain employees, checking of whether a service is applicable at that studio | studioId: int  name: string  clients: unordered\_map<int, Client>  receptionist: Receptionist  photographer: Photographer  administrator: Administrator  supportedServices: vector<ServiceType>  consumedDaily: ConsumedMaterials | Studio(int id, const string n, vector<ServiceType> services)  getName() → string  getId() → int  &getReceptionist() → Receptionist  &getPhotographer() → Photographer  &getAdmin() Administrator  &getClients() → unordered\_map<int, Client>  &getConsumedMaterials() → ConsumedMaterials  offersService(ServiceType s) → bool  getVisibleOrders() → vector<Order> |
| IStudio | Interface | Accessing of Studio functions without having declared Studio | - | &getClients() → unordered\_map<int, Client>  getVisibleOrders() → vector<Order>  offersService(ServiceType s) → bool  getId() → int  &getConsumedMaterials() → ConsumedMaterials |

# 5. Interfaces and Abstractions

<Document all interfaces and abstract classes that support modularity, testing, or future extension. Specify the purpose, key methods, and release when each is planned to appear.  
Examples include IClock, IReportable, and FileService.

|  |  |  |  |
| --- | --- | --- | --- |
| Interface | Purpose | Key Methods | Planned For (Release) |
| IStudio | Allow Receptionist and Photographer to access Studio methods without declaring Studio | &getClients() → unordered\_map<int, Client>  getVisibleOrders() → vector<Order>  offersService(ServiceType s) → bool  getId() → int  &getConsumedMaterials() → ConsumedMaterials |  |

# 6. Function Responsibilities

< Describe the purpose and data flow of each key function or method. This section defines what each function does, what data it uses, and what it produces. >

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Class | Method | Purpose | Input | Output | Notes |
| Order | calculatePrice | Calculate price of service (Add express to normal price) | Service, completion time | Total price, isExpress: bool | - |
| Order | getOrderClient | Return the client’s order | - | orderClient | The output contains all the information needed in the client’s order |
| Order | getOrderPhotographer | Return the photographer’s order | - | orderPhotographer | The output contains all the information needed in the photographer’s order |
| Client | setOrderToClient | Tie an order ID to a client | orderId | - | - |
| Client | getOrder | Get an order for the client based on the ID of the order | Order ID | OrderClient | - |
| Receptionist | createClient | Create a client in the given studio | name | - | - |
| Receptionist | createOrder | Create an order object with given values and return it’s id | ClientId, completion time, type of service | Order ID | - |
| Receptionist | validateClient | Validates the supplied client | Client ID, list of clients | bool | - |
| Receptionist | collectPayment | Mark an order as paid if it is done and not yet paid for | Order ID | - | - |
| Receptionist | checkOrder | Check that supplied order ID is in local studio | Order ID | bool |  |
| Receptionist | generateRevenueReport | Create a daily revenue report file | - | - | - |
| Photographer | processOrder | Processes the order, meaning sets the order as done and uses materials | Order ID, used materials |  |  |
| Photographer | startOrder | Mark an order as taken if it is not already | Order ID |  |  |
| Photographer | makeMaterialsReport | Create a daily used materials report file |  |  |  |
| Administrator | viewConsumedMaterialsReport | View consumed materials report |  |  | Opens the file generated by Photographer |
| Administrator | viewDailyRevenueReport | View daily revenue report |  |  | Opens the file generated by Receptionist |
| Studio | Getters for the studio’s name, id, receptionist photographer, administrator, types of services |  |  |  |  |

# 7. Operation Flow

< Explain the logical flow of operations and how components interact between layers. A diagram or textual flow description should show data movement and control sequence. >

Example: ConsoleUI → Manager → Service → Repository → Report.

1. When a new client comes in, the receptionist needs to create a client object to assign its id and name.

* + The id will be automatically incremented every time a client is created.
  + Receptionist inputs the name.

2. After that the receptionist will create the order (createOrder()).

* + Input the clientId (can query the getLastClientIdUsed()), completionTime and the service type.
  + Creation of the main order struct with the inputs of the receptionist, with an orderId (automatically incremented as well).
  + Create an object order by passing the struct we just created.
  + Calculates the prices and if it is express (<= 24h) and set them into the struct that is inside the object order.
  + It generates two types of orders (client and photographer) and sets them into structs (orderClient and orderPhotographer) into the order object, so we can access the three structs through the same object.
  + Adds the id of the order into the array of Ids inside the client object.
  + Push the order object into the mapping of mainOrders.

NOTE: at this point all the orders are created and accessed through an order object. This order object can be accessed from inside the global mainOrders mapping by knowing the id.

3. Photographer processes the orders (will have view functions to query the orders that need to be done)

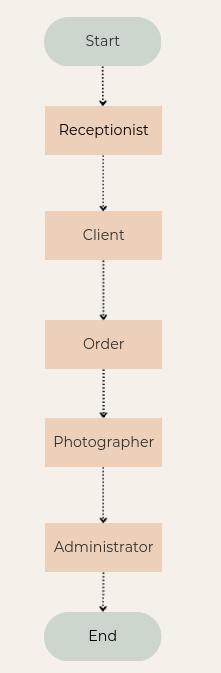
* + Inputs the order id completed and the materials used for that order.
  + Marks the order as done on both structs (orderphotographer and orderdata) inside the order object.
  + There will be a global variable of type struct (consumedDaily), so the materials consumed for that order will be added into that variable (later will help us to make the report for the admin).

4. Once the orders are done, the receptionist can charge the clients and mark the order as paid (will have a getter for the orders done and not paid).

5. At the end of the day the receptionist will need to create a revenue report for the admin (will have a getter for the order paid and not reported).

6. The photographer needs to create his report of daily used materials as well.

* + The same as for the revenue report, but easier on the logic, as we will have the global variable consumedMaterials already summed up. Store that value and reset the variable for the next day.



# 8. Enumerations & Constants

<List all configuration parameters, enumerations, and constants used in the design. >

|  |  |  |
| --- | --- | --- |
| Name | Value / Type | Description |
| enum ServiceType | {PhotoPrinting, FilmDeveloping, PassportPhoto, DigitalEditing} | Supported services |

# 9. Validation Rules & Future Work

< Describe validation logic, exception handling, and planned functionality for future releases. Include placeholder designs and indicate which features are scheduled for Release 3 or beyond. >

|  |  |  |
| --- | --- | --- |
| Rule / Planned Feature | Description | Target Release |

# 10. Traceability Matrix

< Map each requirement from the SRS to its corresponding implementation element in this DLD. This ensures consistency and complete coverage between requirements, design, and code. >

Example:

| **Requirement (SRS)** | **Class / Method (DLD)** |
| --- | --- |
| “As the studio administrator, I want the photographer to have an efficient workflow with as minimal overhead as possible, so that the orders are completed more quickly” | Photographer::startOrder(int orderId)  Photographer::processOrder(int orderId, ConsumedMaterials materials)  Photographer::makeMaterialsReport() |
| “As the receptionist, I want to forward orders with accurate information so that I don’t give out wrong orders.” | Receptionist::createOrder(int clientId, int completionTime, ServiceType service) |
| “As the receptionist, I want to have orders generated within a matter of seconds, so that the customer and photographer don’t have to wait long for them.” | Receptionist::createOrder(int clientId, int completionTime, ServiceType service) |
| “As the receptionist, I want to automatically calculate daily income so that I don’t report wrong numbers.” | Receptionist::generateRevenueReport() |
| “As the photographer, I want to have clear deadlines so that I can complete orders in a sensible sequence.” | Photographer::getFirstOrders() |
| “As the photographer, I want to have an automatic system to calculate daily resource usage so that I don’t have to keep track of it and less mistakes happen.” | Photographer::makeMaterialsReport() |
| “As the client I want to easily place my order so that getting my photo printed doesn’t take me longer than it needs to.” | Receptionist::createClient(string name)  Receptionist::createOrder(int clientId, int completionTime, ServiceType service) |

# 11. Code Structure and File Mapping

< Map all classes and modules to their respective C++ source and header files. This ensures traceability between design and implementation. >

Example table:

| Class | File |
| --- | --- |
| Client | main.c++ |
| Receptionist | main.c++ |
| Order | main.c++ |
| Photographer | main.c++ |
| Administrator | main.c++ |
| Studio | main.c++ |
| IStudio | main.c++ |

# 12. Revision History

<Track document changes across releases. >

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Change Summary | Author |