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.>

# 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.

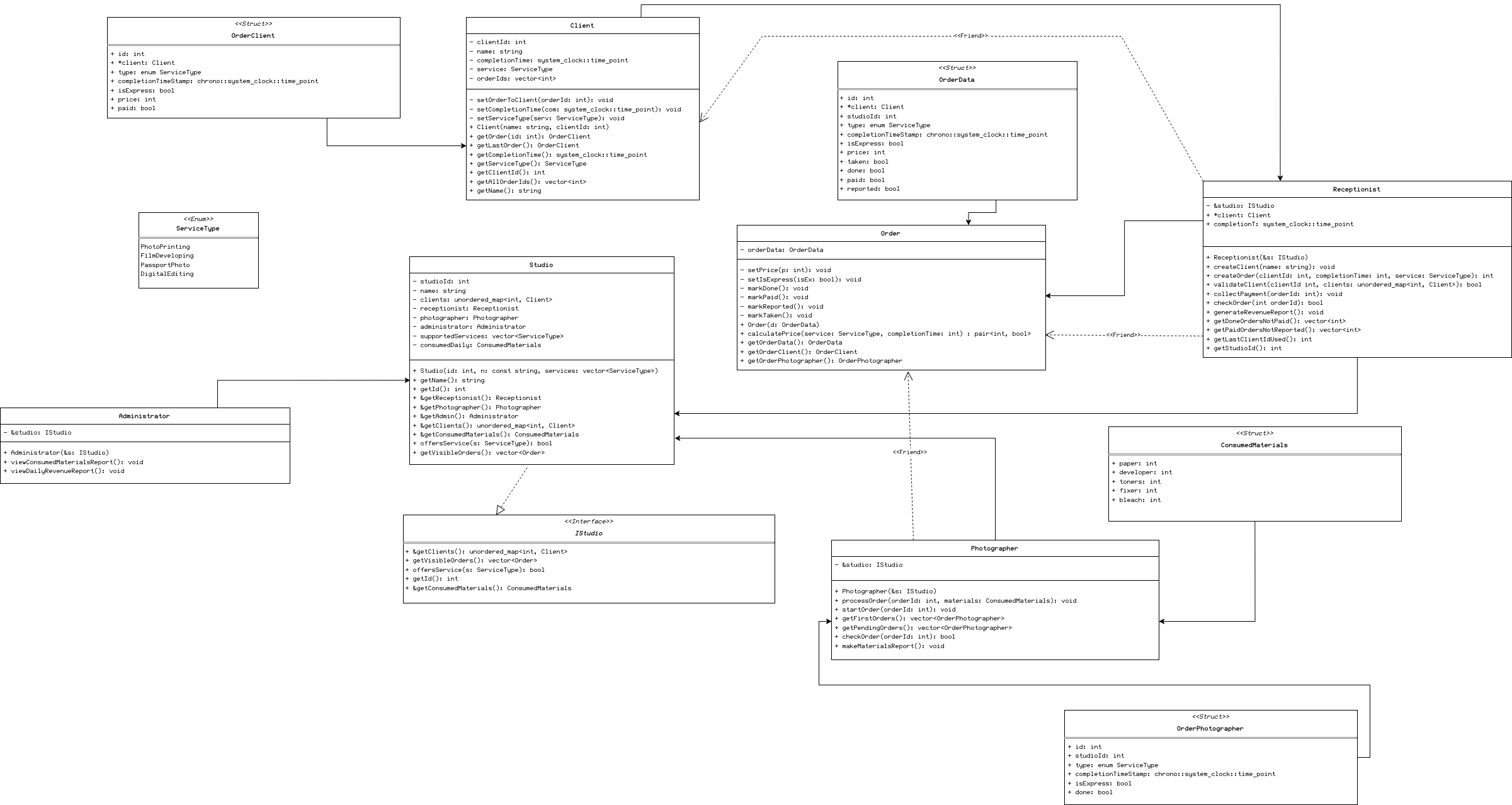
Example:  
- Presentation Layer (UI / ConsoleUI)  
- Logic Layer (Manager, Services)  
- Data Layer (Repositories, FileService)  
Each layer interacts only with the one directly below it. Include a simple schematic or diagram to illustrate dependencies (UI → Logic → Data).>

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 |  |  |  |
| Receptionist | Receptionist |  |  |  |
| Order | Order |  |  |  |
| Photographer |  |  |  |  |
| Administrator |  |  |  |  |
| Studio |  |  |  |  |
| IStudio | Interface |  |  |  |

# 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) |

# 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 |

# 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 autoincremented everytime a client is created.
  + Receptionist inputs the name.

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

* Needs to input the clientId (can query the getLastClientIdUsed()), completionTime and the service type.
* It will create the main order struct with the inputs of the receptionist, with an orderId (autoincremented as well).
* Create an obejct 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 obejct 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. You can access this order object inside 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 above, but easier on the logic, as we will have the global variable consumedMaterials already sum up, we will need to take a snapshot of that 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 |

# 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 a client, I want to place an order" | Client::createOrder() |

# 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 |
| --- | --- |
| Order | order.cpp / order.h |
| Client | client.cpp / client.h |

# 12. Revision History

<Track document changes across releases. >

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