|  |
| --- |
|  |

|  |
| --- |
| REVISION HISTORY |

|  |  |  |  |
| --- | --- | --- | --- |
| **VERSION** | **DATE** | **DESCRIPTION** | **AUTHOR** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |
| --- |
| CONTENTS |

[1 SW System Overview 4](#_Toc209467221)

[1.1 Purpose 4](#_Toc209467222)

[1.2 Scope 4](#_Toc209467223)

[1.3 Use-Case Diagram 5](#_Toc209467224)

[1.4 General Constraints 5](#_Toc209467225)

[1.5 Assumptions and Dependencies 6](#_Toc209467226)

[1.6 Acronyms and Abbreviations 7](#_Toc209467227)

[2 SW Functional Requirements 8](#_Toc209467228)

[2.1 Features / Functions to be Implemented and Acceptance Criteria 8](#_Toc209467229)

[3 SW Non-Functional Requirements 9](#_Toc209467230)

[3.1 Resource Consumption 9](#_Toc209467231)

[3.2 License Issues 9](#_Toc209467232)

[3.2.1 Project Licensing 9](#_Toc209467233)

[3.2.2 Third-Party Libraries & Tools 10](#_Toc209467234)

[3.2.3 Constraints & Compliance 10](#_Toc209467235)

[3.2.4 Future Considerations 10](#_Toc209467236)

[3.3 Coding Standard 10](#_Toc209467237)

[3.4 Modular Design 11](#_Toc209467238)

[3.5 Reliability 12](#_Toc209467239)

[3.6 Portability 13](#_Toc209467240)

[3.7 General Operational Guidelines 13](#_Toc209467241)

[4 SW Design Artifacts 14](#_Toc209467242)

[4.1 CRC Cards (Class–Responsibility–Collaboration) 14](#_Toc209467243)

[4.2 Conceptual UML Diagram (entities & relationships) 15](#_Toc209467244)

# SW System Overview

The Software Requirements Specification defines the functional and non-functional requirements for a Photo Studio Management System (henceforth referred to as PSMS). This system is intended to be used to make the workflow of a small photo studio more efficient by automating order intake, processing, tracking of inventory, and end-of-the-day reporting.

## Purpose

The purpose of PSMS is to improve the efficiency and accuracy of managing photo studio tasks that are traditionally done manually. Tasks such as recording client orders, calculating prices (including surcharges for urgent orders), tracking the use of materials (e.g., photo paper, developer), and generating end-of-day reports.

Problems PSMS solves:

* Inconsistent or incorrect price calculations
* Manual bookkeeping for inventory tracking and revenue that causes manual errors or takes up too much time
* Lost or incomplete order information
* Difficulties keeping track of order status

Intended users:

* Receptionists
* Photographers
* Administrators

PSMS is being developed to create a system for small photo studios with limited technical infrastructure, featuring low-cost automation to reduce workload and minimize errors stemming from human error.

## Scope

In-scope features include:

* Order life cycle:
  + Order creation
  + Calculation of the totals
  + Order storage
* Dual form generation (client form, photographer form)
* Order assignment and processing
* Inventory tracking
* Payment status tracking
* Report generation (revenue report and consumables report)
* Console UI(Role-specific)

Out-of-scope:

* Graphical user interface
* Transaction process
* Media uploads, image processing, and file storage
* Authentication, authorization, and auditing
* Networked access

Benefits of PSMS:

* **Reduced errors**
* **Faster workflows**
* **Better inventory control**
* **Clear accountability**
* **Time savings for staff**
* **Reliable daily bookkeeping**
* **Low-cost, low-risk deployment**
* **Extensible foundation**

## Use-Case Diagram

|  |
| --- |
|  |

## General Constraints

Technical constraints:

* Programming language: C++
* Runtime environment: Windows
* Persistence: Local file-based persistence using plain-text formats
* Memory and storage: Designed to operate comfortably within typical workstation limits. Should handle up to 1000 orders and associated inventory records in memory without swapping
* Not connected to the network
* Single-user assumption: No concurrency control or multi-process locking

Performance constraints:

* Throughput: System should accept and persist a new order in under 200 milliseconds on a typical development machine
* Latency: Common UI operations (create order, list orders, generate report) should respond within 200–500 milliseconds for datasets under 1000 records
* Startup time: Application startup and data load should complete in under 2 seconds for typical datasets
* Scalability thresholds

## Assumptions and Dependencies

Operational assumptions

* Single workstation: The system is operated from a single workstation by studio staff; no simultaneous remote users
* User skills: Users can operate a terminal and follow menu-driven text prompts
* Data volume: Daily order volume is low (tens to low hundreds), and total stored orders remain modest (<500)
* Physical inventory units: Consumables are counted in discrete units
* Cash payments: Payment handling covers local cash transactions only; electronic payments are not supported

Environmental dependencies

* Local file system: Writable local filesystem where the application runs is available and reliable
* Clock availability: Accurate system clock for timestamps and report dating
* Terminal availability: Standard terminal emulator or command prompt on target OS

Software dependencies

* C++ runtime and standard library
* Build tools: Presence of required compilers
* Optional Tools: Diagram tools for documentation, and Git for version control and collaboration

Third-party and external dependencies

* No external APIs
* Optional libraries: If a small helper library is used (e.g., for CSV parsing), it must be permissively licensed and included or documented as a dependency

Data and domain assumptions

* Client identifier simplicity: Client identity is represented by surname only
* Price entry trust: Unit prices are input by staff and trusted; no automated price lookup or tax calculation is included
* Consumable usage estimates: Photographer consumable estimates are approximations and are reconciled when actual usage is recorded

## Acronyms and Abbreviations

|  |  |
| --- | --- |
| **Terms Used** | **Description of terms** |
| **PSMS** | Photo Studio Management System; the name of the software |
| **UI** | User Interface; the screens, dialogs, or console menus through which users interact with the system |
| **OS** | Operating System; the system software that manages hardware and provides services to applications (e.g., Windows, Linux, macOS) |
| **API** | Application Programming Interface; a defined set of functions, methods, or endpoints that allow other software to interact with the application |
| **CSV** | Comma Separated Values; a simple text format for tabular data where fields are separated by commas |
| **CPU** | Central Processing Unit; the primary processor in a computer that executes instructions |
| **RAM** | Random Access Memory; volatile memory used by running programs for fast read/write access |
| **JSON** | JavaScript Object Notation; a lightweight, text-based data format for structured data using key-value pairs and arrays |
| **UML** | Unified Modeling Language; a standardized visual notation for modeling software designs with diagrams |
| **IDE** | Integrated Development Environment; a software application that integrates code editing, building, and debugging tools |
| **RAII** | Resource Acquisition Is Initialization; a C++ idiom that ties resource lifetime to object lifetime so resources are released in destructors |
| **MTTR** | Mean Time To Recover; the average time required to restore a system to operation after a failure |
| **ISO** | International Organization for Standardization; an organization that publishes international technical and industrial standards |
| **I/O** | Input/Output; operations that transfer data between a program and external entities like files, users, or networks |
| UTF-8 | Unicode Transformation Format 8-bit; a variable-length encoding that represents every Unicode character using one to four bytes |
| FK | Foreign Key; a database field that creates a link between two tables by referencing a primary key in another table |

# SW Functional Requirements

## Features / Functions to be Implemented and Acceptance Criteria

|  |  |  |  |
| --- | --- | --- | --- |
| User Story | I want ... (Goal/ Event) | Acceptance Criteria | Validation |
| Client places order  (As a client) | Place an order and receive a receipt so that I can confirm my request | Two order forms are generated (client copy + photographer copy). The price includes a 25% surcharge if express | The system generates two receipts with surname, service type, deadline, urgency, and final price |
| Receptionist records order  (As a receptionist) | Record client details and completion time so that the order is properly transferred | System stores surname, deadline, urgency; orders are retrievable | Database check confirms correct storage; retrieval test ensures orders can be passed forward |
| Photographer processes order  (As a Photographer) | Receive and mark orders as complete so that they are delivered on time | Orders can be marked as “completed.” Completed orders included in daily report | Test case: mark order as complete → verify status change; completed orders appear in report |
| Photographer records consumables  (As a Photographer) | Record consumables usage so that I can report it at the end of the day | Consumables (paper, chemicals, etc.) logged per order; totals aggregated in daily report | Materials log updated per order; daily report totals verified against logged usage |
| Administrator reviews reports  (As an Administrator) | Review revenue and consumables so that I can manage inventory and finances | Daily summary report generated including total revenue and total consumables used | Automated report generation tested; totals match order database and consumables log |

# SW Non-Functional Requirements

## Resource Consumption

|  |  |  |
| --- | --- | --- |
| Metric | Target Value | Validation Method |
| CPU Usage | The application shall not exceed 10% CPU utilization on a mid-range workstation (Intel i5 or AMD equivalent) under normal workload. | Monitor system resources (Task Manager, top/htop) |
| Memory Usage | The system shall operate within 200 MB RAM for up to 500 active orders and associated consumables data. Memory leaks are not permitted. | Runtime profiling and memory leak detection (Valgrind, ASan) |
| Storage per day | Daily files (orders + reports) shall not exceed 5 MB/day under normal workload | Inspect generated CSV/JSON file sizes |
| Response Time | The system shall respond to user actions (e.g., order placement, retrieval) within 1 second, and validation messages within 0.5 seconds under normal workload | Functional testing with stopwatch or automated timers |
| Report Generation | The system shall generate daily revenue/consumables reports within 3 seconds for up to 500 orders/day | Load test with 500 sample orders; measure report generation time |
| Data Retention | The system shall support storage of 2 years of historical records (~3.5 GB in plain text/CSV) without degradation | Long-term accumulation test and file size monitoring |
| Scalability and Limits | The prototype must handle ≥1000 total orders without performance degradation and allow migration to a database backend for larger volumes | Stress testing with 1000+ orders; architecture review for database readiness |

## License Issues

### Project Licensing

* The prototype is developed for educational purposes only (course project)
* No commercial use is intended in the current phase
* A permissive license such as MIT License or Apache 2.0 may be applied if the project is shared publicly (e.g., GitHub)

### Third-Party Libraries & Tools

* The implementation uses standard C++ libraries, which are free and covered by compiler vendor licenses
* If additional libraries are used (e.g., JSON parser, file handling utilities, UML diagramming tools), they must be:
  + Open source
  + Properly documented with attribution in the project report
* Any proprietary libraries or licensed tools (e.g., commercial UML modeling software, paid IDEs) cannot be redistributed with the project

### Constraints & Compliance

* If GPL-licensed libraries are integrated, the project source code may need to be disclosed (copyleft requirement)
* For permissive-licensed libraries (MIT, Apache 2.0, BSD), only attribution and license text inclusion are required
* The team must maintain a License Documentation File (LICENSE.txt) listing all third-party components, their version, and license type

### Future Considerations

* If the system is extended for commercial deployment, compliance with third-party licenses must be reassessed
* Integration with external APIs (e.g., payment providers) will require acceptance of their terms of service and licensing agreements
* Data protection regulations may impose additional legal constraints on handling client information

## Coding Standard

Purpose

Provide consistent, readable, and maintainable C++ code for PSMS that eases review, debugging, and future extension.

Language and Standard

* Use C++17 or later. Prefer modern idioms (RAII, smart pointers, constexpr, range-for)

Naming Conventions

* Files: lowercase with underscores (e.g., order\_manager.cpp, order\_manager.h)
* Classes / Structs: PascalCase (OrderManager, InventoryItem)
* Methods / Functions / Variables: camelCase (createOrder, totalPrice)
* Constants / Macros: ALL\_CAPS\_WITH\_UNDERSCORES (DEFAULT\_TAX\_RATE)
* Namespaces: short lowercase (psms)

Formatting and Layout

* Indent with 4 spaces (no tabs)
* Max line length: 100 characters
* One blank line between logical blocks; two between function definitions
* Opening brace on same line for functions and control blocks:
  + Correct:
    - void foo() {

// ...

}

* Use consistent vertical spacing for readability

Header Files

* Header contents: class/struct declarations, inline small functions
* Implementation (logic) in .cpp files

Comments and Documentation

* Use Doxygen-style comments for public APIs
* Prefer self-documenting names; use comments for intent, non-obvious algorithms, and invariants
* Avoid commented-out dead code in the main branch; use version control instead

Error Handling

* Use exceptions for unrecoverable or unexpected errors in core logic; catch at top-level CLI loop to present user-friendly messages
* Use std::optional or error codes for expected failure paths where appropriate

Resource Management

* Prefer std::unique\_ptr and std::shared\_ptr; avoid raw owning pointers
* Manage file handles and other resources with RAII wrappers

Unit Tests

* Write unit tests for core business logic (pricing, inventory deduction, state transitions).
* Use a lightweight test framework or simple hand-rolled test harness if adding frameworks is undesired
* Static Analysis and Linting
* Run clang-tidy / cppcheck where available
* Adopt a simple clang-format configuration and apply consistently

Version Control

* Use Git. Commit often with clear messages. Keep feature branches small and merge via pull requests.

## Modular Design

Architectural Goals

* Modularity, separation of concerns, extensibility, and testability

Layered Components

* UI Layer (Console): Menu/navigation, input validation, formatting of reports and forms. Should contain no business logic
* Application / Use-Case Layer: Orchestrates use-cases (createOrder, assignOrder, generateReport). Implements workflows and enforces state transitions
* Domain / Business Layer: Core domain classes and logic (Order, Inventory, PricingEngine, ReportGenerator). Pure business rules, fully unit-testable
* Utilities: Logging, time utilities, validation helpers, CSV parsing

Extensibility

* Design PricingEngine as strategy-like component allowing new pricing rules (discounts, taxes)
* Persistence interface should allow adding database-backed implementations without changing business logic

Cohesion and Coupling

* Maximize cohesion inside modules; minimize coupling between modules
* Communicate via well-defined data structures rather than shared global state

Testability

* Inject dependencies (e.g., Persistence, Clock) via constructor parameters for easy mocking in tests

## Reliability

Availability Targets

* Prototype targeted for local single-user availability; aim for mean-time-to-recover (MTTR) measured in minutes for file corruption or app crash

Error Detection and Reporting

* Validate all user inputs with clear, actionable error messages
* Detect and report file I/O errors with context (filename, operation, suggested remedies)
* Log critical errors to a timestamped log file

Fault Tolerance and Recovery

* Use atomic write pattern for persistence: write to temporary file then rename to replace original
* Maintain a simple rotating backup (keep last N versions, e.g., N=3) of primary data files
* On startup, perform integrity checks on data files; if corrupted, attempt to recover from backup and notify the user

Transactional Integrity

* For multi-step operations (e.g., process order → deduct inventory → mark paid), ensure either full completion or rollback of in-memory state and do not persist partial updates
* Persist critical state promptly after successful completion of operations

Input Validation and Defensive Programming

* Validate ranges (quantities >= 0, prices >= 0), date formats, and enumerated values
* Reject invalid state transitions (e.g., cannot mark as Completed unless status is Processing)

Logging and Auditing

* Log key events with ISO 8601 timestamps: order creation, assignment, processing, payment, inventory updates, and errors
* Keep logs human-readable and rotate logs daily or when size limit reached

Testing for Reliability

* Include unit tests for error paths
* Add integration tests that simulate file failure and recovery scenarios where feasible

## Portability

Target Platforms

* Windows (Command Prompt / PowerShell)

Portability Requirements

* Use only standard C++ library features and OS-agnostic APIs for file I/O and time
* Handle line-ending differences when reading/writing files
* Use UTF-8 encoding for all textual files

Build and Distribution

* Document build steps for each platform in README
* Avoid platform-specific system calls; isolate any unavoidable OS-specific code behind small abstraction layers

File Paths and Permissions

* Accept a configurable data directory via command-line argument or environment variable
* Use portable path manipulation (std::filesystem).
* Check and report file/directory permissions and create data directories if missing.

## General Operational Guidelines

* Scalability: Prototype must support up to 1000 orders; design must allow database migration for larger datasets
* Robustness: System must maintain a consistent state even after abnormal termination (e.g., incomplete file writes)
* Ease of Use: User interface (console) shall be simple, menu-driven, and avoid unnecessary complexity
* Maintainability: Code must follow defined coding standards (naming, documentation, RAII, exceptions)
* Modifiability: Business rules (e.g., surcharge %) must be configurable via external settings, not hard-coded

# SW Design Artifacts

## CRC Cards (Class–Responsibility–Collaboration)

|  |  |  |
| --- | --- | --- |
| **Class** | **Responsibilities** | **Collaborators** |
| Client | * Place orders * Provide surname * Provide the deadline * Receive completed results | Receptionist  Order |
| Receptionist | * Record client details and deadline * Create orders * Pass orders * Calculate payment * Submit reports * View orders | Client  Order  Photographer  Studio Administrator |
| Photographer | * Receive and view order * Process orders * Mark completion * Record consumables usage * Generate daily materials report | Order  Studio Administrator  Consumable  Receptionist |
| Order | * Store client name and order\_id * Due-date * Price calculation * Order status * Store photographer user\_id | Client  Receptionist  Photographer |
| Consumable | * Track materials and quantities * Store material\_instance\_id * Update stock | Photographer  Studio Administrator |
| Studio Administrator | * Review revenue and consumables reports * Manage inventory and finances * Create and delete users * View orders | Receptionist  Photographer  Consumable |

## Conceptual UML Diagram (entities & relationships)