Part 2 – Implementation Report
Point-of-Sale (POS) System

Group XXX: Member A Member B Member C

January 11, 2026

Contents

1 Introduction

This report documents the implementation phase of our Point-of-Sale (POS) system, developed as the second part of the *Software Engineering Practice* assignment. The project demonstrates the application of Object-Oriented (OO) analysis, design and implementation techniques, supported by modern version-control practices with Git and GitHub.

Core objectives were:

- Provide a working POS application capable of **processing sales** and **handling returns**.[1]
- Apply a **three-layer architecture** (UI, Service, Domain) with clear separation of concerns.
- Employ OO principles (abstraction, encapsulation, inheritance, polymorphism) and relevant design patterns (Service Layer, Domain Model).
- Maintain full team collaboration through feature branches, pull-requests and documented Git history.

2 System Overview

2.1 Use Case Model

Figure **??** shows the Use Case Diagram for the POS system. The two primary use cases required by the assignment are **Process Sale** and **Handle Returns**.

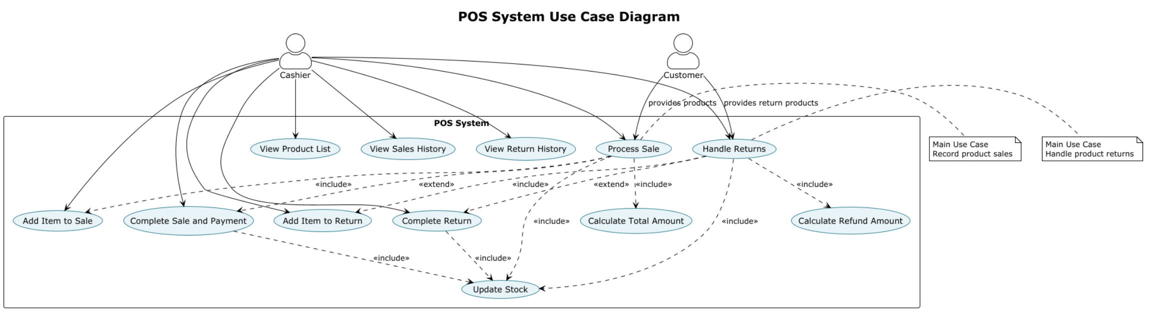


Figure 1: Use Case Diagram of the POS System

2.2 Architecture

The system adopts a three-layer logical architecture. Figure **??** presents the package/module organisation used in implementation, which helps separate UI, service logic, and domain model concerns.

[1]Both CLI and GUI front-ends are available.

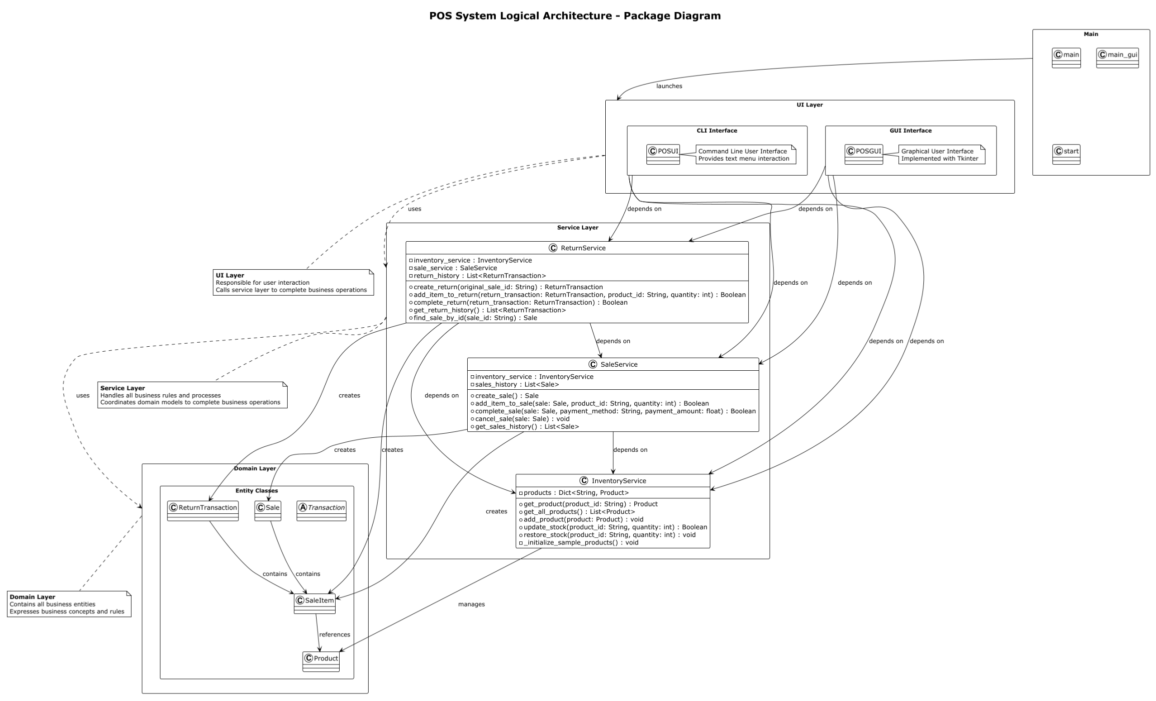

POS System Logical Architecture - Package Diagram
Main
main
main_gui
start
launches
UI Layer
CLI Interface
POSUI
Command Line User Interface
Provides text menu interaction
GUI Interface
POSGUI
Graphical User Interface
Implemented with Tkinter
uses
depends on
Service Layer
ReturnService
inventory_service : InventoryService
sale_service : SaleService
return_history : List<ReturnTransaction>
create_return(original_sale_id: String) : ReturnTransaction
add_item_to_return(return_transaction: ReturnTransaction, product_id: String, quantity: int) : Boolean
complete_return(return_transaction: ReturnTransaction) : Boolean
get_return_history() : List<ReturnTransaction>
find_sale_by_id(sale_id: String) : Sale
SaleService
inventory_service : InventoryService
sales_history : List<Sale>
create_sale() : Sale
add_item_to_sale(sale: Sale, product_id: String, quantity: int) : Boolean
complete_sale(sale: Sale, payment_method: String, payment_amount: float) : Boolean
cancel_sale(sale: Sale) : void
get_sales_history() : List<Sale>
InventoryService
products : Dict<String, Product>
get_product(product_id: String) : Product
get_all_products() : List<Product>
add_product(product: Product) : void
update_stock(product_id: String, quantity: int) : Boolean
restore_stock(product_id: String, quantity: int) : void
_initialize_sample_products() : void
UI Layer
Responsible for user interaction
Calls service layer to complete business operations
Service Layer
Handles all business rules and processes
Coordinates domain models to complete business operations
creates
manages
Domain Layer
Entity Classes
ReturnTransaction
Sale
Transaction
contains
SaleItem
references
Product
Domain Layer
Contains all business entities
Expresses business concepts and rules

Figure 2: Logical Architecture Package Diagram

2.3 Domain Model

Key business entities appear in the domain layer (Figure **??**). They are implemented in the `domain/` package and form the backbone of the application logic.

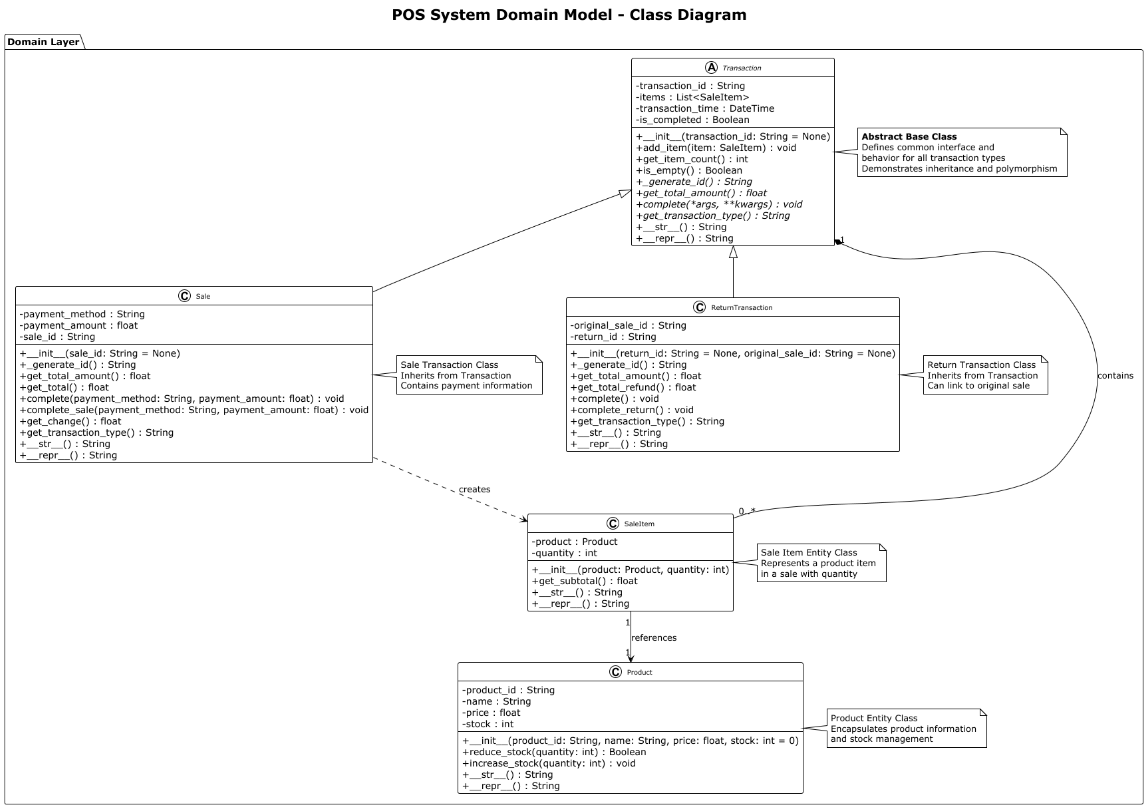

POS System Domain Model - Class Diagram
Domain Layer
Transaction
-transaction_id : String
-items : List<SaleItem>
-transaction_time : DateTime
-is_completed : Boolean
+__init__(transaction_id: String = None)
+add_item(item: SaleItem) : void
+get_item_count() : int
+is_empty() : Boolean
+_generate_id() : String
+get_total_amount() : float
+complete(*args, **kwargs) : void
+get_transaction_type() : String
+__str__() : String
+__repr__() : String
Abstract Base Class
Defines common interface and
behavior for all transaction types
Demonstrates inheritance and polymorphism
Sale
-payment_method : String
-payment_amount : float
-sale_id : String
+__init__(sale_id: String = None)
+_generate_id() : String
+get_total_amount() : float
+get_total() : float
+complete(payment_method: String, payment_amount: float) : void
+complete_sale(payment_method: String, payment_amount: float) : void
+get_change() : float
+get_transaction_type() : String
+__str__() : String
+__repr__() : String
Sale Transaction Class
Inherits from Transaction
Contains payment information
ReturnTransaction
-original_sale_id : String
-return_id : String
+__init__(return_id: String = None, original_sale_id: String = None)
+_generate_id() : String
+get_total_amount() : float
+get_total_refund() : float
+complete() : void
+complete_return() : void
+get_transaction_type() : String
+__str__() : String
+__repr__() : String
Return Transaction Class
Inherits from Transaction
Can link to original sale
1
contains
0..*
creates
SaleItem
-product : Product
-quantity : int
+__init__(product: Product, quantity: int)
+get_subtotal() : float
+__str__() : String
+__repr__() : String
Sale Item Entity Class
Represents a product item
in a sale with quantity
1
references
1
Product
-product_id : String
-name : String
-price : float
-stock : int
+__init__(product_id: String, name: String, price: float, stock: int = 0)
+reduce_stock(quantity: int) : Boolean
+increase_stock(quantity: int) : void
+__str__() : String
+__repr__() : String
Product Entity Class
Encapsulates product information
and stock management

Figure 3: Domain Model Class Diagram

3 System Development

3.1 Tasks Completed

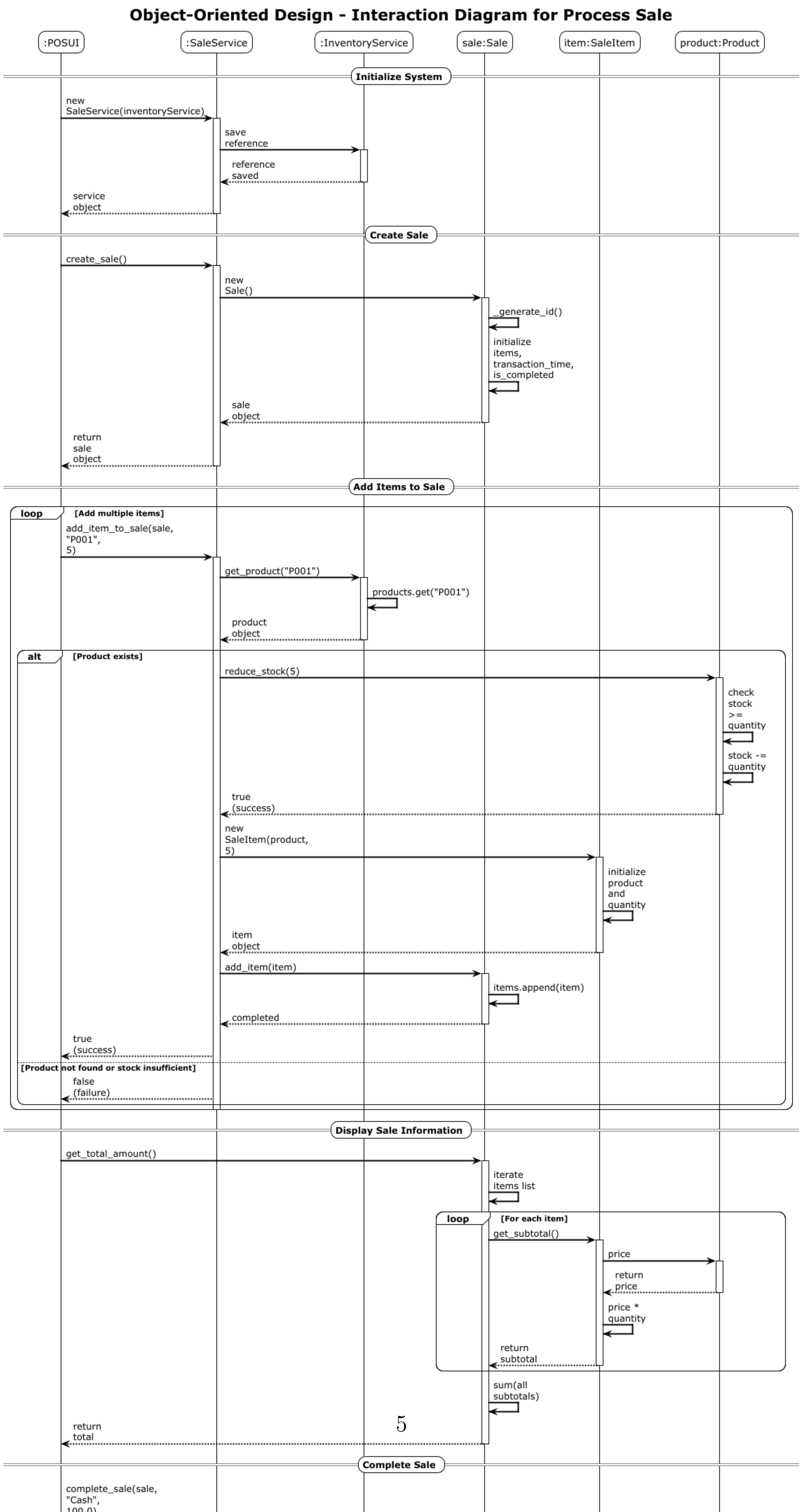
Table **??** summarises the functional and documentation artefacts delivered.

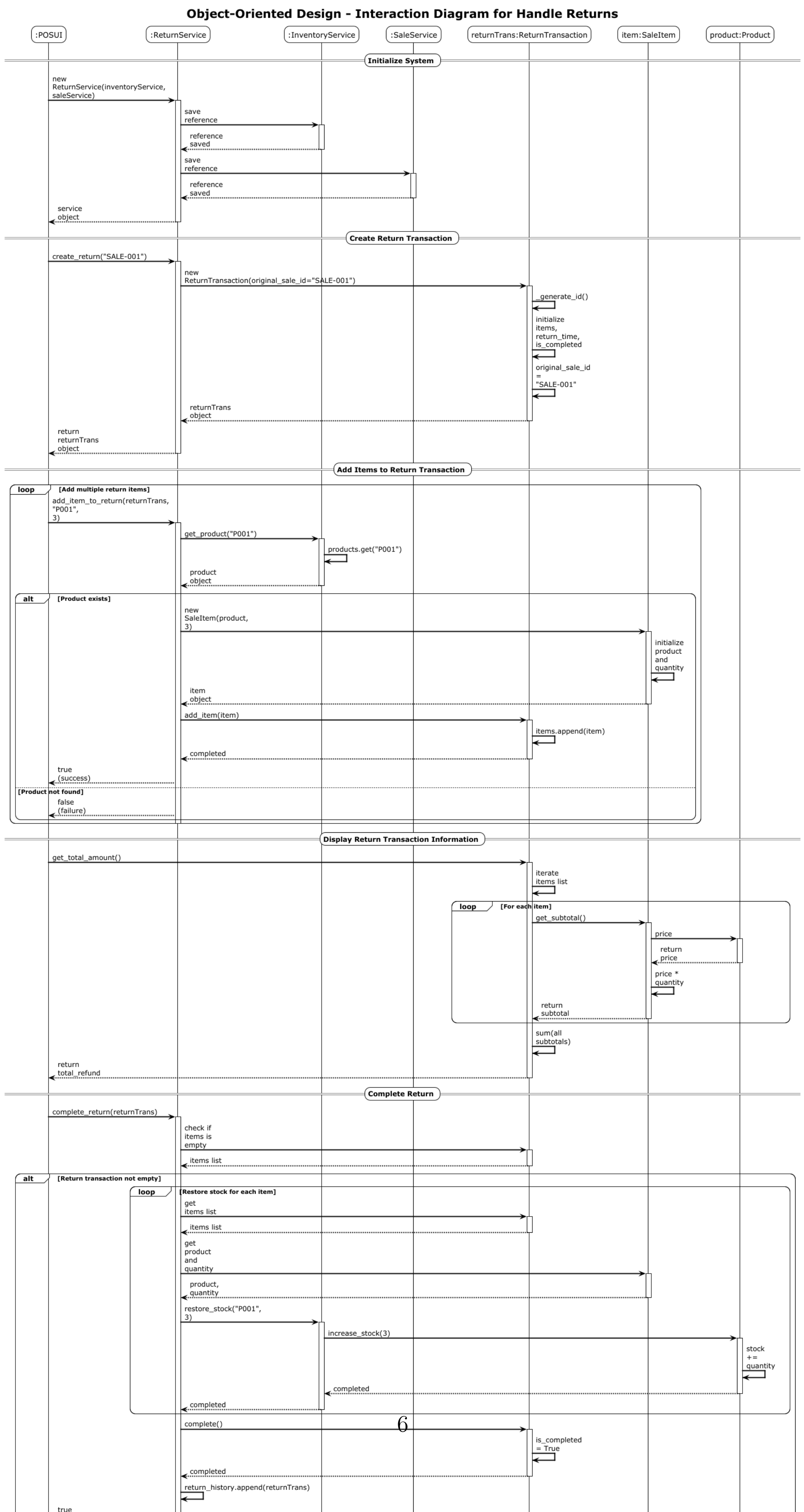
Category	**Deliverables**
Project Structure	Three-layer package skeleton (`domain/`, `service/`, `ui/`)
Domain Layer	`Product`, `Sale`, `SaleItem`, `ReturnTransaction`
Service Layer	`InventoryService`, `SaleService`, `ReturnService`
UI Layer	CLI (`pos_ui.py`), GUI (`pos_gui.py`) built with Tkinter
Core Use-Cases	Process Sale, Handle Returns (end-to-end)
Persistence	In-memory store; interface prepared for future DB layer
Tests	`test_pos_system.py` unit / integration tests
UML Artefacts	Use-Case diagram, SSDs, Domain Class diagram, Package diagram, Interaction diagrams (PlantUML)
Documentation	README, ARCHITECTURE, PROJECT_SUMMARY, this Report

Table 1: Summary of completed development tasks

3.2 Interaction Diagrams

To illustrate the dynamic behaviour of key use cases, we include two sequence diagrams generated with PlantUML.


Object-Oriented Design - Interaction Diagram for Process Sale
:POSUI
:SaleService
:InventoryService
sale:Sale
item:SaleItem
product:Product
Initialize System
new SaleService(inventoryService)
save reference
reference saved
service object
Create Sale
create_sale()
new Sale()
_generate_id()
initialize items, transaction_time, is_completed
sale object
return sale object
Add Items to Sale
loop
[Add multiple items]
add_item_to_sale(sale, "P001", 5)
get_product("P001")
products.get("P001")
product object
alt
[Product exists]
reduce_stock(5)
check stock >= quantity
stock -= quantity
true (success)
new SaleItem(product, 5)
initialize product and quantity
item object
add_item(item)
items.append(item)
completed
true (success)
[Product not found or stock insufficient]
false (failure)
Display Sale Information
get_total_amount()
iterate items list
loop
[For each item]
get_subtotal()
price
return price
price * quantity
return subtotal
sum(all subtotals)
return total
Complete Sale
complete_sale(sale, "Cash",


Object-Oriented Design - Interaction Diagram for Handle Returns
:POSUI
:ReturnService
:InventoryService
:SaleService
returnTrans:ReturnTransaction
item:SaleItem
product:Product
Initialize System
new ReturnService(inventoryService, saleService)
save reference
reference saved
save reference
reference saved
service object
Create Return Transaction
create_return("SALE-001")
new ReturnTransaction(original_sale_id="SALE-001")
_generate_id()
initialize items, return_time, is_completed
original_sale_id = "SALE-001"
returnTrans object
return returnTrans object
Add Items to Return Transaction
loop
[Add multiple return items]
add_item_to_return(returnTrans, "P001", 3)
get_product("P001")
products.get("P001")
product object
alt
[Product exists]
new SaleItem(product, 3)
initialize product and quantity
item object
add_item(item)
items.append(item)
completed
true (success)
[Product not found]
false (failure)
Display Return Transaction Information
get_total_amount()
iterate items list
loop
[For each item]
get_subtotal()
price
return price
price * quantity
return subtotal
sum(all subtotals)
return total_refund
Complete Return
complete_return(returnTrans)
check if items is empty
items list
alt
[Return transaction not empty]
loop
[Restore stock for each item]
get items list
items list
get product and quantity
product, quantity
restore_stock("P001", 3)
increase_stock(3)
stock += quantity
completed
completed
complete()
is_completed = True
completed
return_history.append(returnTrans)
true

4 Implementation Snippets

Rather than embedding the entire code-base, representative excerpts are included using `lstinputlisting`. Full source is available in the repository.

4.1 Domain Layer

```python
"""
Transaction Abstract Base Class

"""

from abc import ABC, abstractmethod
from datetime import datetime
from typing import List
from domain.sale_item import SaleItem


class Transaction(ABC):
    """




    """

    def __init__(self, transaction_id: str = None):
        """



        Args:
            transaction_id:                  IDNone
        """
        self.transaction_id = transaction_id or self._generate_id()
        self.items: List[SaleItem] = []
        self.transaction_time = datetime.now()
        self.is_completed = False

    @abstractmethod
    def _generate_id(self) -> str:
        """
                                  ID
                                  ID

        Returns:
            str:    ID
        """
        pass

    def add_item(self, item: SaleItem):
        """


        Args:
            item:
        """
        self.items.append(item)

    @abstractmethod
    def get_total_amount(self) -> float:
        """



        Returns:
            float:
        """
        pass

    @abstractmethod
    def complete(self, *args, **kwargs):
        """



        Args:
            *args:
            **kwargs:
        """
        pass

    def get_item_count(self) -> int:
        """


        Returns:
            int:
        """
        return len(self.items)

    def is_empty(self) -> bool:
        """


        Returns:
            bool:             True
        """
        return len(self.items) == 0

    @abstractmethod
    def get_transaction_type(self) -> str:
        """


        Returns:
            str:
        """
        pass

    def __str__(self):
        """                                                     """
        items_str = "\n".join(f"  - {item}" for item in self.items)
        status = "Completed" if self.is_completed else "In Progress"
        return f"{self.get_transaction_type()} {self.transaction_id} ({status})\n{items_str}"

    def __repr__(self):
        """            """
        return f"{self.__class__.__name__}(transaction_id='{self.transaction_id}', items={len(self.items)})"
```

Listing 1: Transaction Abstract Class

```python
"""
Sale Entity Class
          Transaction

"""

from datetime import datetime
from domain.sale_item import SaleItem
from domain.transaction import Transaction


class Sale(Transaction):
    """

              Transaction


    """

    def __init__(self, sale_id: str = None):
        """


        Args:
            sale_id:                  IDNone
        """
        #
        super().__init__(sale_id)
        # Sale specific attributes
        self.payment_method = None
        self.payment_amount = 0.0

        # Compatibility aliases
        self.sale_id = self.transaction_id  # old attribute name
        self.sale_time = self.transaction_time  # maintain old attribute

    def _generate_id(self) -> str:
        """
                                 ID

        Returns:
            str:    ID
        """
        return f"SALE-{datetime.now().strftime('%Y%m%d%H%M%S')}"

    def get_total_amount(self) -> float:
        """


        Returns:
            float:
        """
        return sum(item.get_subtotal() for item in self.items)

    def get_total(self) -> float:
        """


        Returns:
            float:
        """
        return self.get_total_amount()

    def complete(self, payment_method: str, payment_amount: float):
        """


        Args:
            payment_method:
            payment_amount:
        """
        self.payment_method = payment_method
        self.payment_amount = payment_amount
        self.is_completed = True

    def complete_sale(self, payment_method: str, payment_amount: float):
        """


        Args:
            payment_method:
            payment_amount:
        """
        self.complete(payment_method, payment_amount)

    def get_change(self) -> float:
        """


        Returns:
            float:
        """
        if self.is_completed:
            return self.payment_amount - self.get_total_amount()
        return 0.0

    def get_transaction_type(self) -> str:
        """


        Returns:
            str:
        """
        return "Sale"

    def __str__(self):
        """                                                       """
        items_str = "\n".join(f"  - {item}" for item in self.items)
        status = "Completed" if self.is_completed else "In Progress"
        return f"Sale {self.transaction_id} ({status})\n{items_str}\nTotal: ${self.get_total_amount():.2f}"

    def __repr__(self):
        """              """
        return f"Sale(transaction_id='{self.transaction_id}', items={len(self.items)}, total={self.get_total_amount()})"
```

Listing 2: Sale Entity Class

4.2 Service Layer

```
"""
Inventory Management Service
"""

from typing import Dict, Optional
from domain.product import Product

```

```python

class InventoryService:
    """Inventory management service class"""

    def __init__(self):
        """Initialize inventory service"""
        self.products: Dict[str, Product] = {}
        self._initialize_sample_products()

    def _initialize_sample_products(self):
        """Initialize sample products"""
        sample_products = [
            Product("P001", "Apple", 5.50, 100),
            Product("P002", "Banana", 3.80, 80),
            Product("P003", "Milk", 12.00, 50),
            Product("P004", "Bread", 8.50, 60),
            Product("P005", "Egg", 15.00, 40),
        ]
        for product in sample_products:
            self.products[product.product_id] = product

    def get_product(self, product_id: str) -> Optional[Product]:
        """
        Get product by ID

        Args:
            product_id: Product ID

        Returns:
            Product: Product object, or None if not found
        """
        return self.products.get(product_id)

    def get_all_products(self) -> list[Product]:
        """
        Get all products

        Returns:
            list: List of all products
        """
        return list(self.products.values())

    def add_product(self, product: Product):
        """
        Add product

        Args:
            product: Product object
        """
        self.products[product.product_id] = product

    def update_stock(self, product_id: str, quantity: int) -> bool:
        """
        Update stock (decrease)

        Args:
            product_id: Product ID
            quantity: Quantity (positive to decrease, negative to increase)

        Returns:
            bool: Whether update was successful
        """
        product = self.get_product(product_id)
        if product:
            if quantity > 0:
                return product.reduce_stock(quantity)
            else:
                product.increase_stock(-quantity)
                return True
        return False

    def restore_stock(self, product_id: str, quantity: int):
        """
        Restore stock (increase)

        Args:
            product_id: Product ID
            quantity: Quantity to restore
        """
        product = self.get_product(product_id)
        if product:
            product.increase_stock(quantity)
```

Listing 3: Inventory Service

```python
"""
Sale Service
"""

from typing import List
from domain.sale import Sale
from domain.sale_item import SaleItem
from domain.product import Product
from service.inventory_service import InventoryService


class SaleService:
    """Sale service class"""

    def __init__(self, inventory_service: InventoryService):
        """
        Initialize sale service

        Args:
            inventory_service: Inventory service object
        """
        self.inventory_service = inventory_service
        self.sales_history: List[Sale] = []

    def create_sale(self) -> Sale:
        """
        Create new sale

        Returns:
            Sale: New sale object
        """
        return Sale()

    def add_item_to_sale(self, sale: Sale, product_id: str,
quantity: int) -> bool:
        """
        Add item to sale

        Args:
            sale: Sale object
            product_id: Product ID
            quantity: Quantity

        Returns:
            bool: Whether addition was successful
        """
        product = self.inventory_service.get_product(product_id)
        if not product:
            return False

        if not product.reduce_stock(quantity):
            return False

        item = SaleItem(product, quantity)
        sale.add_item(item)
        return True

    def complete_sale(self, sale: Sale, payment_method: str,
payment_amount: float) -> bool:
        """
        Complete sale

        Args:
            sale: Sale object
            payment_method: Payment method
            payment_amount: Payment amount

        Returns:
            bool: Whether completion was successful
        """
        total = sale.get_total()
        if payment_amount < total:
            return False

        sale.complete_sale(payment_method, payment_amount)
        self.sales_history.append(sale)
        return True

    def cancel_sale(self, sale: Sale):
        """
        Cancel sale (restore stock)

        Args:
            sale: Sale object
        """
        for item in sale.items:
            self.inventory_service.restore_stock(item.product.product_id, item.quantity)

    def get_sales_history(self) -> List[Sale]:
        """
        Get sales history

        Returns:
            List: List of sales history
        """
        return self.sales_history.copy()
```

Listing 4: Sale Service

5 Git Command History

Team collaboration was fully tracked in GitHub. A condensed commit log covering all branches is embedded below (generated on January 11, 2026):

```
% Run in repo root:
%   git log --graph --oneline --all --decorate --since=2025-01-01 > docs/report/git_h
```

6 Conclusion

The project successfully delivered a functional POS system demonstrating complete OO lifecycle coverage—from requirements modelling through to working software and documentation. Key take-aways include:

- The three-layer architecture proved effective in separating UI, service logic and domain concerns.
- PlantUML integrated smoothly into the documentation workflow, enabling living UML diagrams.

- Collective code ownership was facilitated by strict branch discipline and frequent merges, resulting in minimal integration conflicts.

Future work may integrate persistent storage, extend reporting capabilities and introduce fine-grained user authentication.