D780 - Task 1 - Attempt 1

Silver Alcid - #011933973

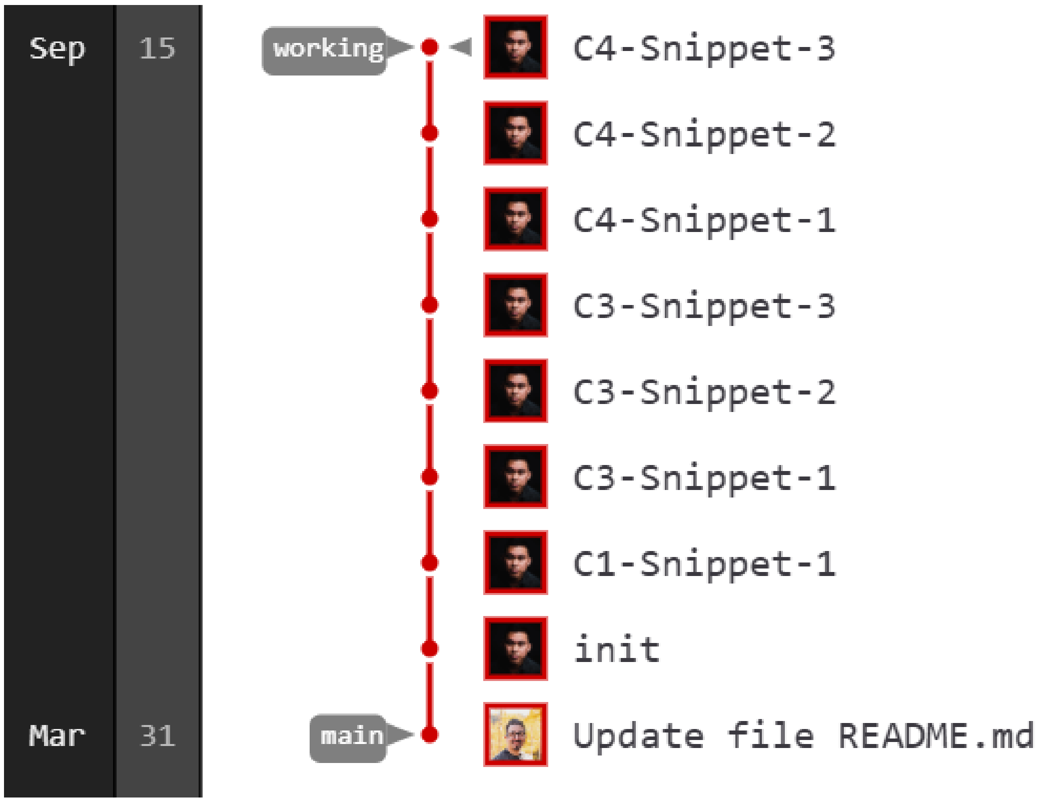
9/15/2025

Software Architecture & Design – Task 1

A3. Submit a copy of the GitLab repository URL in the "Comments to Evaluator" section

https://gitlab.com/wgu-gitlab-environment/student-repos/aalcid/d780-software-architecture-and-design/-/tree/f68cfe6af6c1fefb0e60c75382e16db33516cde7/

A4. Submit a copy of the repository branch history retrieved from your repository, which must include the commit messages and dates



```
Sep 15   working ► C4-Snippet-3
                   C4-Snippet-2
                   C4-Snippet-1
                   C3-Snippet-3
                   C3-Snippet-2
                   C3-Snippet-1
                   C1-Snippet-1
                   init
Mar 31   main    ► Update file README.md
```

B1. Identify the design pattern present in each of the three provided code snippets found in the attached "Code Snippets" text file.

Snippet #1: Cart Component

Snippet #1 (Cart Component) implements the Singleton pattern. By overriding _new_ and caching the sole instance in the class attribute _instance, every construction attempt returns the same object, so all callers share a single items list. This provides a global, centralized cart state across the application without passing references around. The tradeoffs are the typical ones for singletons. A hidden global state that can complicate testing and concurrency, no synchronization here, and tighter coupling to a single global lifecycle.

Snippet #2: Payment Component

Snippet #2 (Payment Component) implements a Simple Factory (static factory method) pattern. PaymentProcessorFactory.get_processor(method) encapsulates the object-creation logic and returns a concrete processor (CreditCardProcessor or PayPalProcessor) based on a runtime selection. This separates "which class to instantiate" from the client's payment workflow, improving substitutability and keeping client code closed to concrete types. It is still a simple factory rather than a full Abstract Factory. Extending to new methods requires modifying the factory's conditional, but the clients remain decoupled from constructor details.

Snippet #3: Inventory Component

Snippet #3 (Inventory Component) implements the Strategy pattern. The Inventory class is composed with a strategy object whose update_stock method defines the algorithm for changing inventory; different concrete strategies (AddStockStrategy, RemoveStockStrategy) can be injected to vary behavior at runtime without altering Inventory. This demonstrates "composition over inheritance" and avoids scattering conditionals inside Inventory for each update policy. The cost is an extra level of indirection and more objects to manage, but it yields clean separation of policy from mechanism and supports easy extension with new update strategies.

B2. List the criteria used to identify each of the three design patterns in part B1.

Snippet #1: Cart Component

Singleton:

- Overrides _new_ to control construction and return a cached instance.
- Maintains a class-level _instance used to enforce a single object identity.
- Performs lazy initialization of shared mutable state (items) on first instantiation.
- Ensures all callers obtain the same instance (global, centralized state), not independent objects.

Snippet #2: Payment Component

Simple Factory:

- Uses a separate factory type (PaymentProcessorFactory) dedicated to object creation.
- Exposes a single static selector (get_processor(method)) that encapsulates branching.
- Selects a concrete class based on a runtime token ("credit_card", "paypal") instead of clients instantiating directly.
- Returns objects adhering to a common protocol (process_payment) via duck typing, decoupling clients from concrete types.

- Lacks creator subclassing or product families, distinguishing it from Factory Method/Abstract Factory.

Snippet #3: Inventory Component

Strategy:

- Defines a context (Inventory) composed with an injected strategy at construction time.
- Delegates the variable behavior to strategy.update_stock(...) at a single call site.
- Provides multiple interchangeable concrete strategies (AddStockStrategy, RemoveStockStrategy) with a uniform interface.
- Replaces conditional logic in the context with polymorphic behavior, enabling algorithm substitution at runtime.

B3. Explain how the application of the criteria presented in part B2 supports the identifications made in part B1.

Snippet #1: Cart Component

The identification is supported by its control over instance creation through an overridden _new_ method and a class-level cache (_instance) that returns the same object on every construction attempt. This mechanism enforces a single, globally shared identity—every client receives the same Cart instance—while performing lazy initialization of shared mutable state (items) at first creation. These structural and behavioral cues align directly with the Singleton pattern’s defining properties: one instance, a global access point, and construction control embedded in the class itself.

Snippet #2: Payment Component

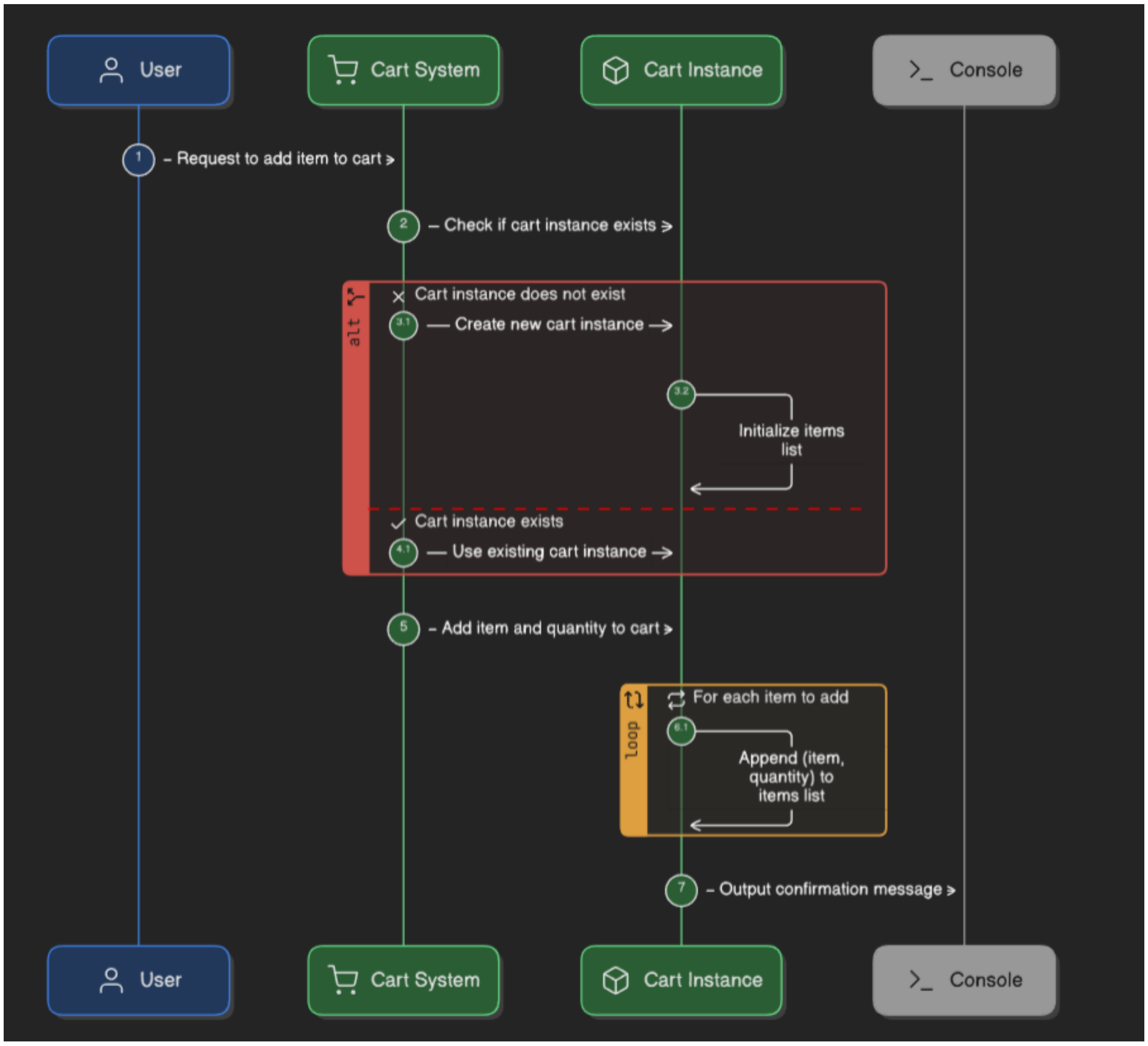
The use of a dedicated creator class (PaymentProcessorFactory) with a static selector method (get_processor(method)) that encapsulates branching logic substantiates the Simple Factory identification. Clients supply a token (e.g., "credit_card", "paypal") rather than instantiating concrete classes directly, and they receive an object that conforms to a common operational protocol (process_payment). Because there is no hierarchy of creator subclasses and no families of related products, this structure maps to a Simple (static) Factory rather than Factory Method or Abstract Factory, confirming the pattern choice.

Snippet #3: Inventory Component

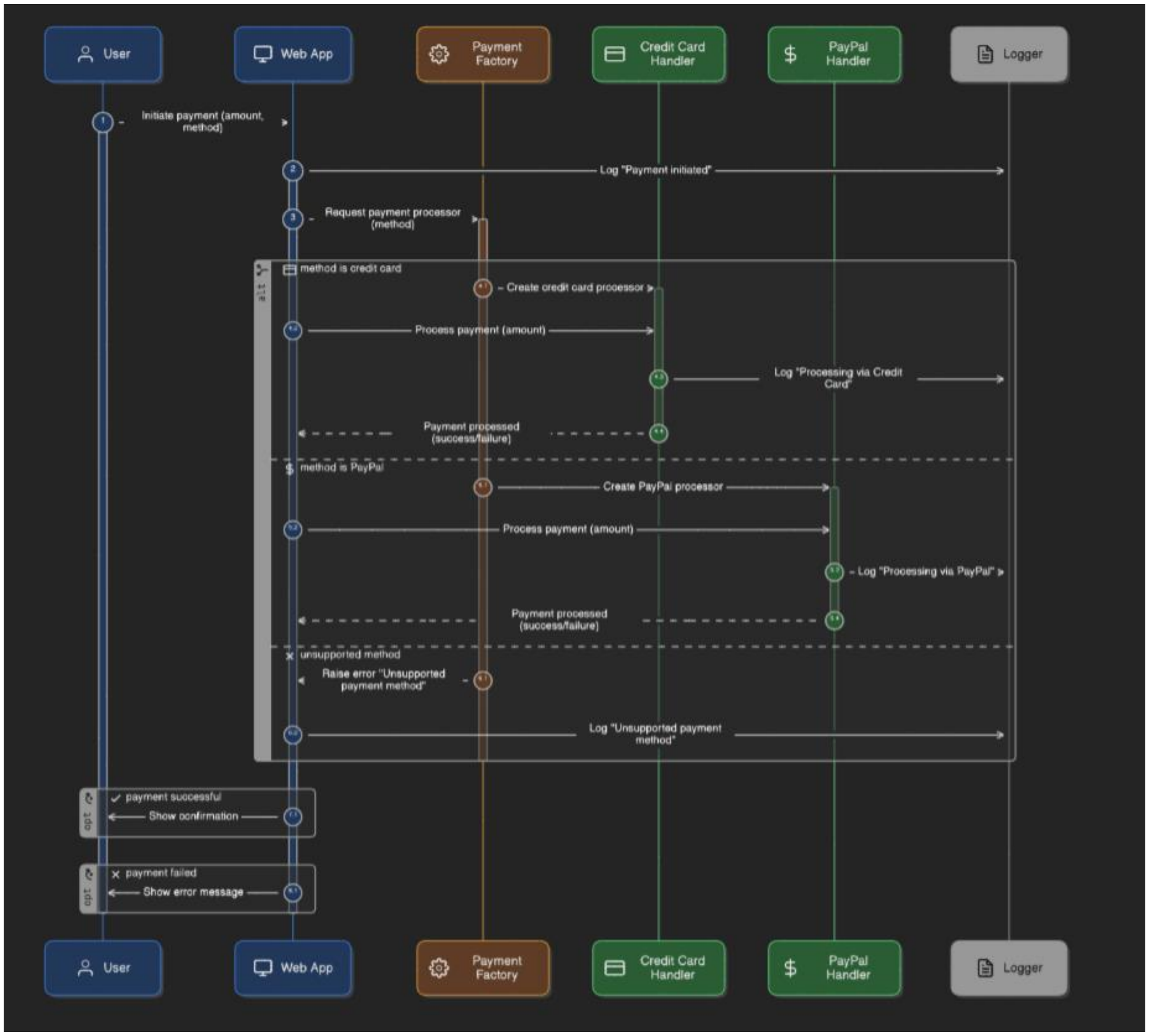
Inventory composes a strategy object and delegates the variable operation to strategy.update_stock(...), demonstrating the classic context–strategy relationship. The presence of multiple interchangeable implementations (AddStockStrategy, RemoveStockStrategy) with the same interface enables behavior substitution at runtime without modifying the Inventory class or introducing conditionals inside it. This separation of policy (how stock changes) from mechanism (when and where updates occur) is the hallmark of the Strategy pattern, validating the identification.

B4. Create a supporting UML class diagram for each of the three design patterns. The class diagrams must contain all relevant objects and relationships.

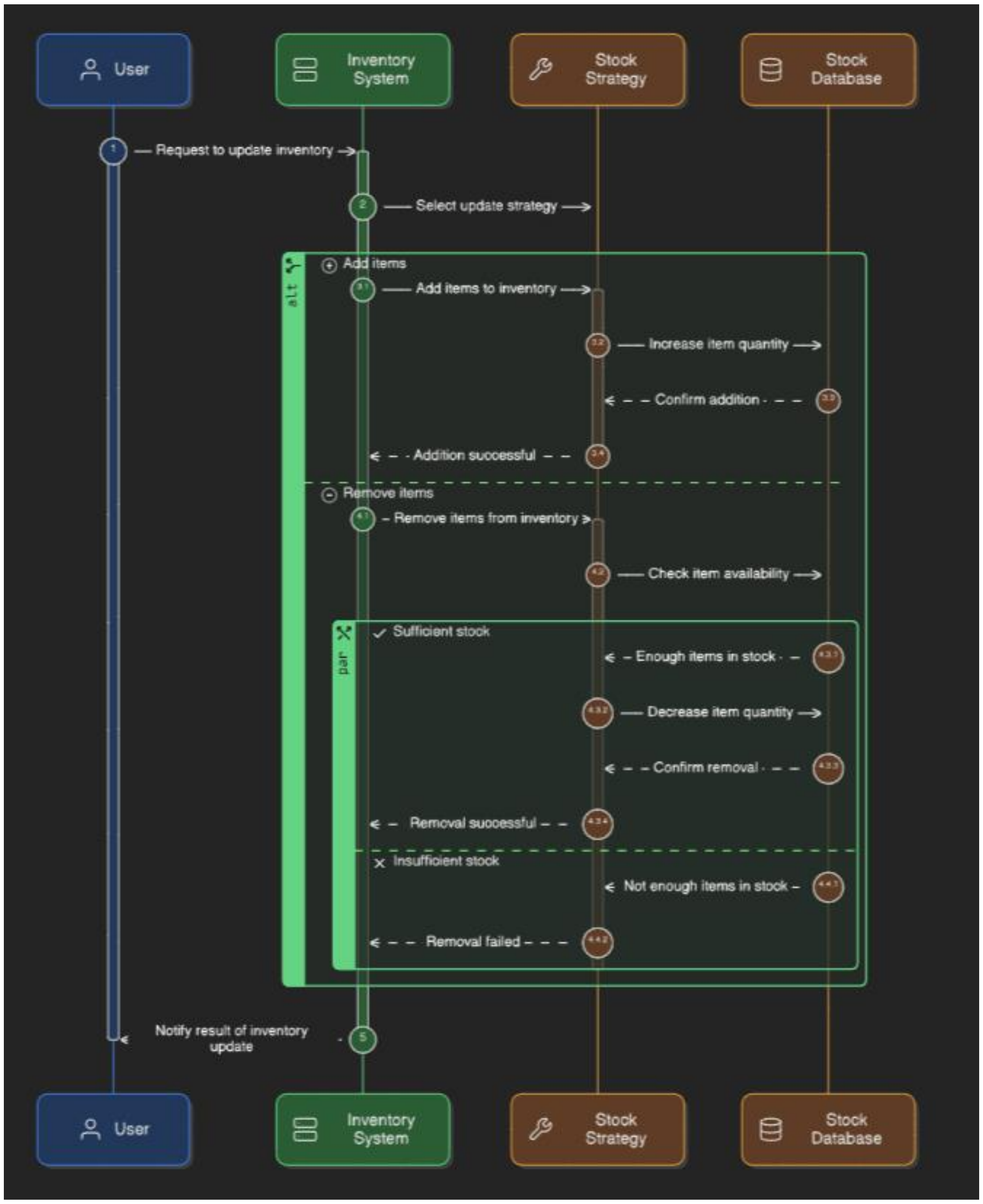
Snippet #1: Cart Component


User
Cart System
Cart Instance
Console
1 – Request to add item to cart
2 – Check if cart instance exists
alt
Cart instance does not exist
3.1 — Create new cart instance
3.2 Initialize items list
Cart instance exists
4.1 — Use existing cart instance
5 – Add item and quantity to cart
loop
For each item to add
6.1 Append (item, quantity) to items list
7 – Output confirmation message
User
Cart System
Cart Instance
Console

Snippet #2: Payment Component


User
Web App
Payment Factory
Credit Card Handler
PayPal Handler
Logger
Initiate payment (amount, method)
Log "Payment initiated"
Request payment processor (method)
alt
method is credit card
Create credit card processor
Process payment (amount)
Log "Processing via Credit Card"
Payment processed (success/failure)
method is PayPal
Create PayPal processor
Process payment (amount)
Log "Processing via PayPal"
Payment processed (success/failure)
unsupported method
Raise error "Unsupported payment method"
Log "Unsupported payment method"
opt
payment successful
Show confirmation
opt
payment failed
Show error message

Snippet #3: Inventory Component



C1. Discuss why the original design pattern present in each of the three provided code snippets is inefficient or inferior based on the provided business scenario.

Snippet #1: Cart Component

The Singleton is inferior for a grocery system because it collapses all users into one shared cart, which breaks the domain requirement that each shopper has an isolated cart per session or account. It is not thread safe and can corrupt state under concurrent requests. It does not work in a distributed deployment because each process creates its own “single” instance, which leads to inconsistent behavior across nodes. It hides global state, which makes unit tests flaky and order dependent. It also blocks common features such as multiple carts per user, saved or draft carts, cart merge after login, and durable persistence with auditing, since the cart lives only in memory without a stable identity.

Snippet #2: Payment Component

The Simple Factory is inferior because it centralizes a growing conditional that must be edited for every new payment method, which violates the open–closed principle and hurts extensibility. It provides no explicit interface, so clients rely on duck typing and discover errors only at runtime. The static factory makes dependency injection and environment configuration difficult, which complicates testing with fakes and switching credentials between sandbox and production. Cross-cutting concerns such as idempotency, retries, fraud checks, and observability either bloat the factory or get duplicated in each processor. As payment options and regional gateways expand, this approach becomes hard to maintain and scale.

Snippet #3: Inventory Component

The current Strategy usage is inferior because it models mutually different operations as a single injected policy, which forces callers to swap strategies just to perform add versus remove in the same workflow. Real inventory needs coordinated operations such as receipts, allocations, picks, returns, and shrink, often within one transaction. The strategies directly mutate a dictionary and bypass invariants, so stock can go negative and there is no validation, locking, or transactional control. There is no persistence boundary, audit trail, or eventing to notify other subsystems such as replenishment or pricing. Domain rules become scattered across small classes, which reduces cohesion and makes composite operations harder to implement correctly.

C2. Identify a different, more appropriate design pattern from the list below for each of the three provided code snippets based on the provided business scenario and justify the identification of this pattern.

Snippet #1: Cart Component

Repository pattern would be the best choice. A CartRepository provides create, load, save, and merge operations keyed by a cart or user identifier, which gives each shopper an isolated cart with a durable identity. This removes the hidden global state of a singleton, works in distributed deployments because state is stored in a backing store, and supports transactions for concurrent updates. It also enables dependency injection and test doubles, which improves maintainability and unit testing. The repository boundary makes persistence, auditing, and recovery explicit and keeps the Cart model focused on behavior rather than storage concerns.

Snippet #2: Payment Component

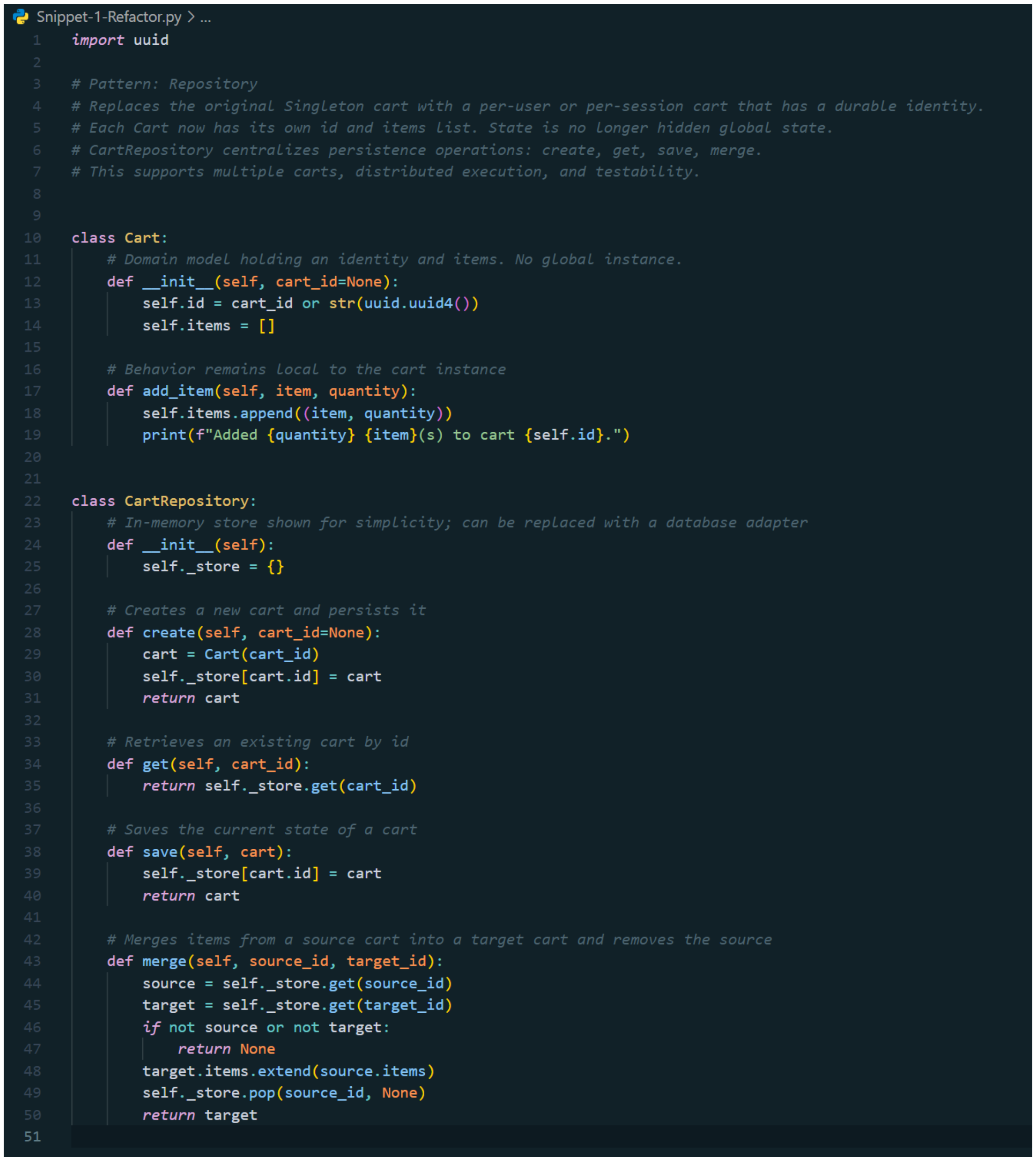
Strategy pattern would be the best choice. Define a PaymentProcessor interface with a process_payment operation and implement a concrete strategy per method such as credit card or PayPal. Select the strategy at runtime through configuration or a registry, not by editing a central conditional. This satisfies open–closed in spirit without using em dashes by allowing new processors to be added without modifying client code. It enables dependency injection, clearer testing with fakes, and separate handling of gateway specific rules. Cross cutting concerns such as logging, retries, or idempotency can be layered later using Decorator if needed, while Strategy keeps the selection and execution of payment logic clean and extensible.

Snippet #3: Inventory Component

Façade pattern would be the best choice. Provide an InventoryService façade that exposes cohesive operations such as add_stock, remove_stock, reserve, and release. Internally it coordinates validation, invariants, transactions, and persistence via an InventoryRepository and can publish domain events to other subsystems. This avoids swapping a single injected strategy to change behavior mid workflow and gives one consistent entry point that enforces rules like nonnegative stock and atomic updates. The façade simplifies client code, centralizes policy, and supports scalability and correctness in a grocery inventory context where operations must be coordinated and auditable.

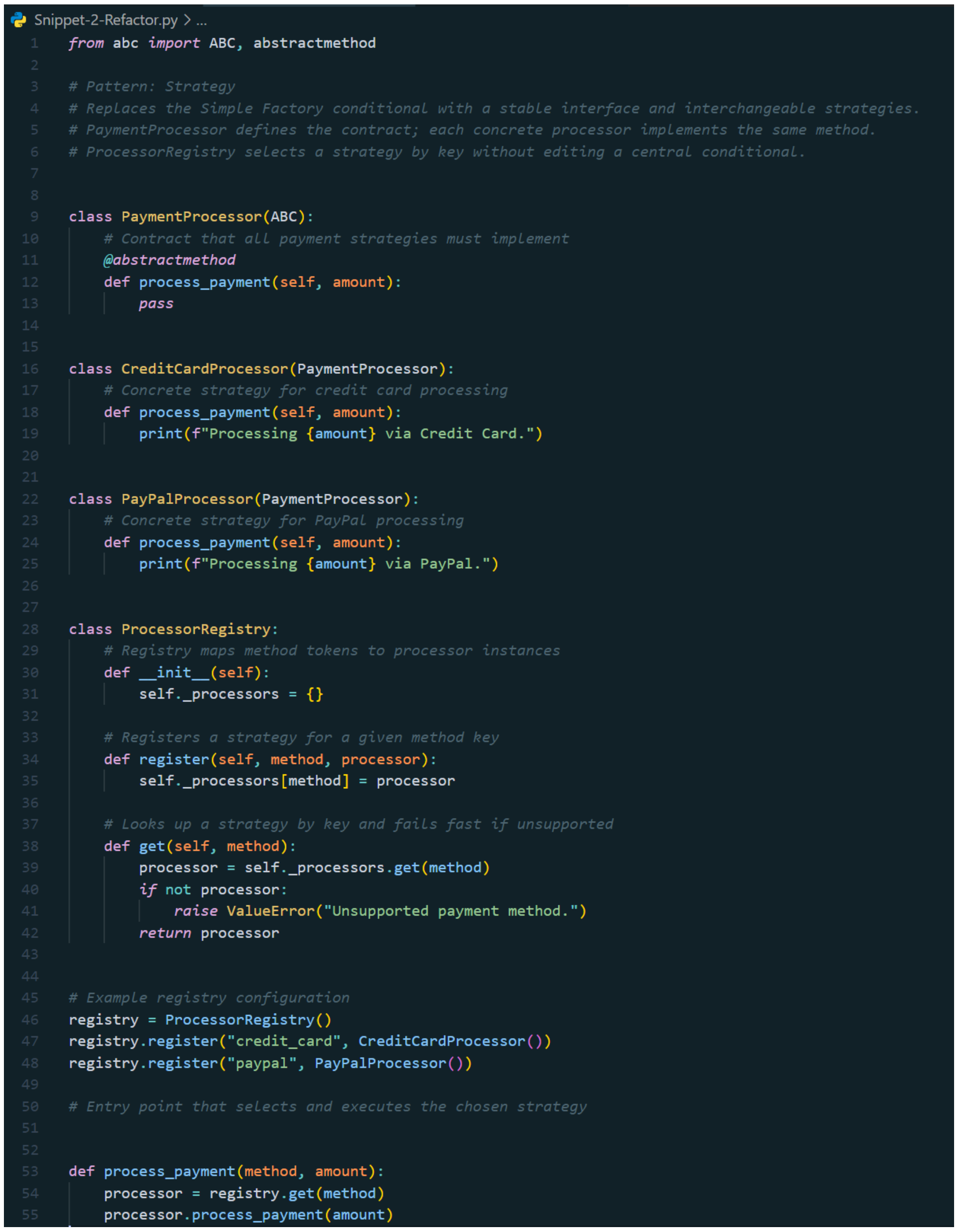
C3 & C4. Update the provided code for each snippet to implement the more appropriate design pattern identified in part C2. Comment the updated code for each snippet to explain all changes from the original code and include comments that explain the new design pattern for each snippet.

Snippet #1: Cart Component


Snippet-1-Refactor.py > ...

```python
import uuid

# Pattern: Repository
# Replaces the original Singleton cart with a per-user or per-session cart that has a durable identity.
# Each Cart now has its own id and items list. State is no longer hidden global state.
# CartRepository centralizes persistence operations: create, get, save, merge.
# This supports multiple carts, distributed execution, and testability.


class Cart:
    # Domain model holding an identity and items. No global instance.
    def __init__(self, cart_id=None):
        self.id = cart_id or str(uuid.uuid4())
        self.items = []

    # Behavior remains local to the cart instance
    def add_item(self, item, quantity):
        self.items.append((item, quantity))
        print(f"Added {quantity} {item}(s) to cart {self.id}.")


class CartRepository:
    # In-memory store shown for simplicity; can be replaced with a database adapter
    def __init__(self):
        self._store = {}

    # Creates a new cart and persists it
    def create(self, cart_id=None):
        cart = Cart(cart_id)
        self._store[cart.id] = cart
        return cart

    # Retrieves an existing cart by id
    def get(self, cart_id):
        return self._store.get(cart_id)

    # Saves the current state of a cart
    def save(self, cart):
        self._store[cart.id] = cart
        return cart

    # Merges items from a source cart into a target cart and removes the source
    def merge(self, source_id, target_id):
        source = self._store.get(source_id)
        target = self._store.get(target_id)
        if not source or not target:
            return None
        target.items.extend(source.items)
        self._store.pop(source_id, None)
        return target

```

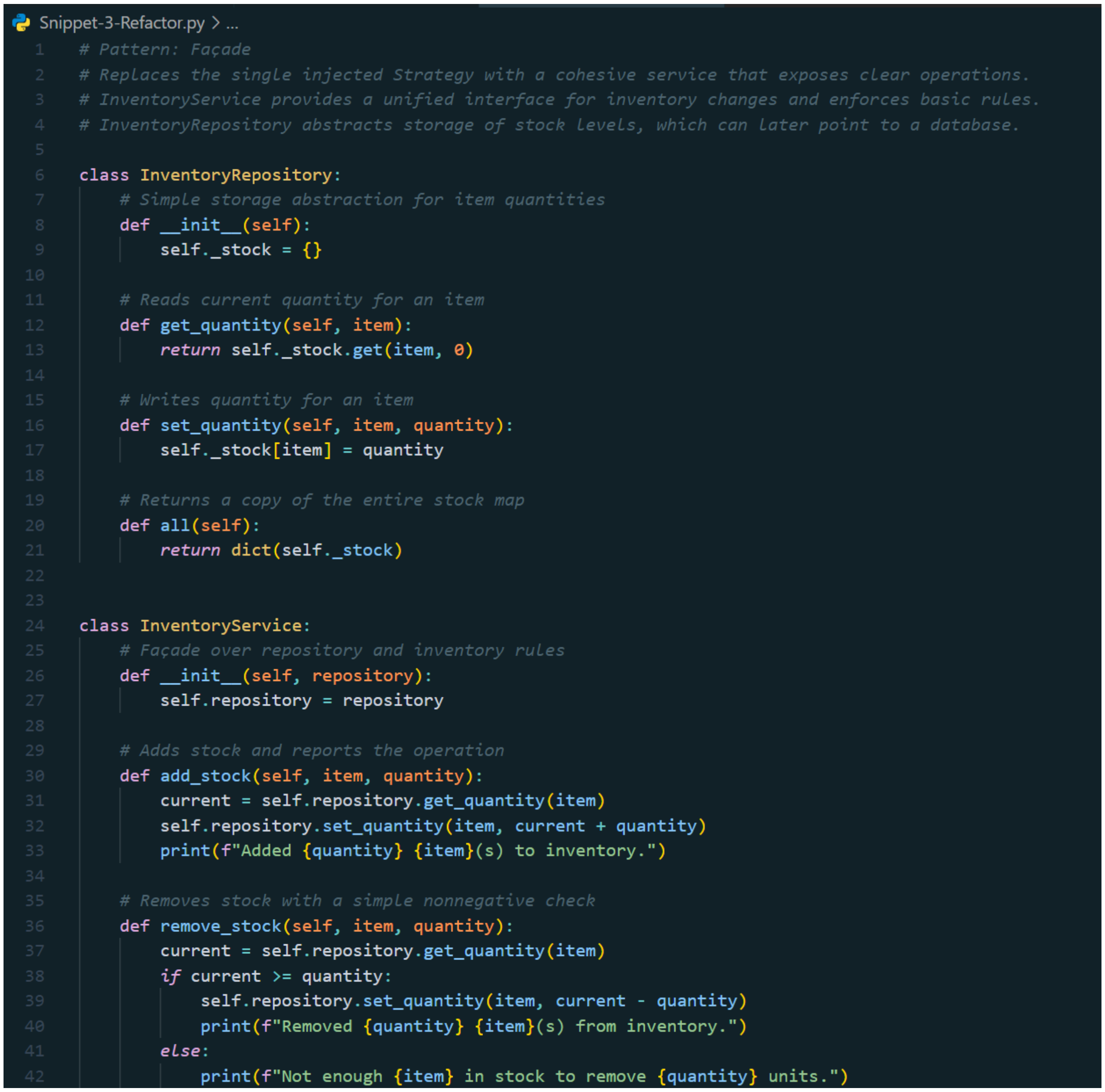
Snippet #2: Payment Component



Snippet-2-Refactor.py > ...

```python
from abc import ABC, abstractmethod

# Pattern: Strategy
# Replaces the Simple Factory conditional with a stable interface and interchangeable strategies.
# PaymentProcessor defines the contract; each concrete processor implements the same method.
# ProcessorRegistry selects a strategy by key without editing a central conditional.


class PaymentProcessor(ABC):
    # Contract that all payment strategies must implement
    @abstractmethod
    def process_payment(self, amount):
        pass


class CreditCardProcessor(PaymentProcessor):
    # Concrete strategy for credit card processing
    def process_payment(self, amount):
        print(f"Processing {amount} via Credit Card.")


class PayPalProcessor(PaymentProcessor):
    # Concrete strategy for PayPal processing
    def process_payment(self, amount):
        print(f"Processing {amount} via PayPal.")


class ProcessorRegistry:
    # Registry maps method tokens to processor instances
    def __init__(self):
        self._processors = {}

    # Registers a strategy for a given method key
    def register(self, method, processor):
        self._processors[method] = processor

    # Looks up a strategy by key and fails fast if unsupported
    def get(self, method):
        processor = self._processors.get(method)
        if not processor:
            raise ValueError("Unsupported payment method.")
        return processor


# Example registry configuration
registry = ProcessorRegistry()
registry.register("credit_card", CreditCardProcessor())
registry.register("paypal", PayPalProcessor())

# Entry point that selects and executes the chosen strategy


def process_payment(method, amount):
    processor = registry.get(method)
    processor.process_payment(amount)
```

Snippet #3: Inventory Component



Snippet-3-Refactor.py > ...

```python
# Pattern: Façade
# Replaces the single injected Strategy with a cohesive service that exposes clear operations.
# InventoryService provides a unified interface for inventory changes and enforces basic rules.
# InventoryRepository abstracts storage of stock levels, which can later point to a database.

class InventoryRepository:
    # Simple storage abstraction for item quantities
    def __init__(self):
        self._stock = {}

    # Reads current quantity for an item
    def get_quantity(self, item):
        return self._stock.get(item, 0)

    # Writes quantity for an item
    def set_quantity(self, item, quantity):
        self._stock[item] = quantity

    # Returns a copy of the entire stock map
    def all(self):
        return dict(self._stock)


class InventoryService:
    # Façade over repository and inventory rules
    def __init__(self, repository):
        self.repository = repository

    # Adds stock and reports the operation
    def add_stock(self, item, quantity):
        current = self.repository.get_quantity(item)
        self.repository.set_quantity(item, current + quantity)
        print(f"Added {quantity} {item}(s) to inventory.")

    # Removes stock with a simple nonnegative check
    def remove_stock(self, item, quantity):
        current = self.repository.get_quantity(item)
        if current >= quantity:
            self.repository.set_quantity(item, current - quantity)
            print(f"Removed {quantity} {item}(s) from inventory.")
        else:
            print(f"Not enough {item} in stock to remove {quantity} units.")
```

D. Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

No sources or references were needed or used.