Assignment 1: State

Software Design Patterns

Student ID: XXXXXXXXX
Name: Colby Frison
Course: CS 4203 - Fall 2025

October 5, 2025

Contents

1 Part A: System Architecture Breakdown

1.1 Selected System: Online Shopping Platform

I have chosen to analyze an **Online Shopping Platform** similar to a simplified version of Amazon. This system allows shoppers to browse items, add them to a shopping cart, and complete purchases online. Sellers can list their items and manage inventory.

Note on Controller Pattern: The controller pattern is applied at the class design level (shown in Part C's Class Diagram with the ShoppingController), not at the high-level architecture/subsystem level described here in Part A. Part A focuses on the overall system structure and how major subsystems interact, while Part C shows how specific classes implement patterns like Controller and Expert within those subsystems.

1.2 Overall Architecture

The Online Shopping Platform follows a **Three-Tier Layered Architecture**:

1. **Presentation Layer** - Handles all user interactions through web/mobile interfaces
2. **Business Logic Layer** - Contains the core application logic for item listings, inventory, pricing, and order processing
3. **Data Layer** - Manages persistent storage of items, sellers, shoppers, and orders in databases

This layered approach keeps different parts of the system separate, where each layer handles its own job and only talks to the layers next to it. The presentation layer can't directly touch the database—it has to go through the business logic layer first, which makes sure all business rules are followed before any data gets changed.

1.3 Three Main Subsystems

1.3.1 User Interface Subsystem

Responsibilities:

- Display item catalog with search and filter capabilities
- Handle user authentication (login/registration for both shoppers and sellers)
- Manage shopping cart display and checkout forms
- Provide seller dashboard for listing management
- Render order confirmation and history pages

1.3.2 Item & Inventory Management Subsystem

Responsibilities:
- Track available item quantities across multiple sellers
- Update stock levels when orders are placed
- Manage item information (title, description, price, seller)
- Handle seller inventory updates and new listings
- Provide item search and recommendation functionality

1.3.3 Order Processing Subsystem

Responsibilities:
- Create and validate shopper orders
- Calculate totals including taxes and shipping
- Process payment transactions
- Generate order confirmations
- Update order status (pending, shipped, delivered)
- Coordinate seller fulfillment

1.4 Subsystem Interactions

User Interface ↔ Item & Inventory Management:
- UI requests item catalog data for display
- UI sends search queries and receives filtered results
- UI retrieves real-time stock availability for each item
- Seller interface submits new listings and inventory updates

User Interface ↔ Order Processing:
- UI submits shopping cart contents to create orders
- UI receives order confirmation details
- UI queries order history for shopper accounts

Order Processing ↔ Item & Inventory Management:
- Order Processing checks stock availability before order confirmation
- Order Processing requests inventory reduction when order is placed
- Inventory Management notifies Order Processing of out-of-stock items
- Order Processing may reserve items temporarily during checkout

1.5 Block Diagram

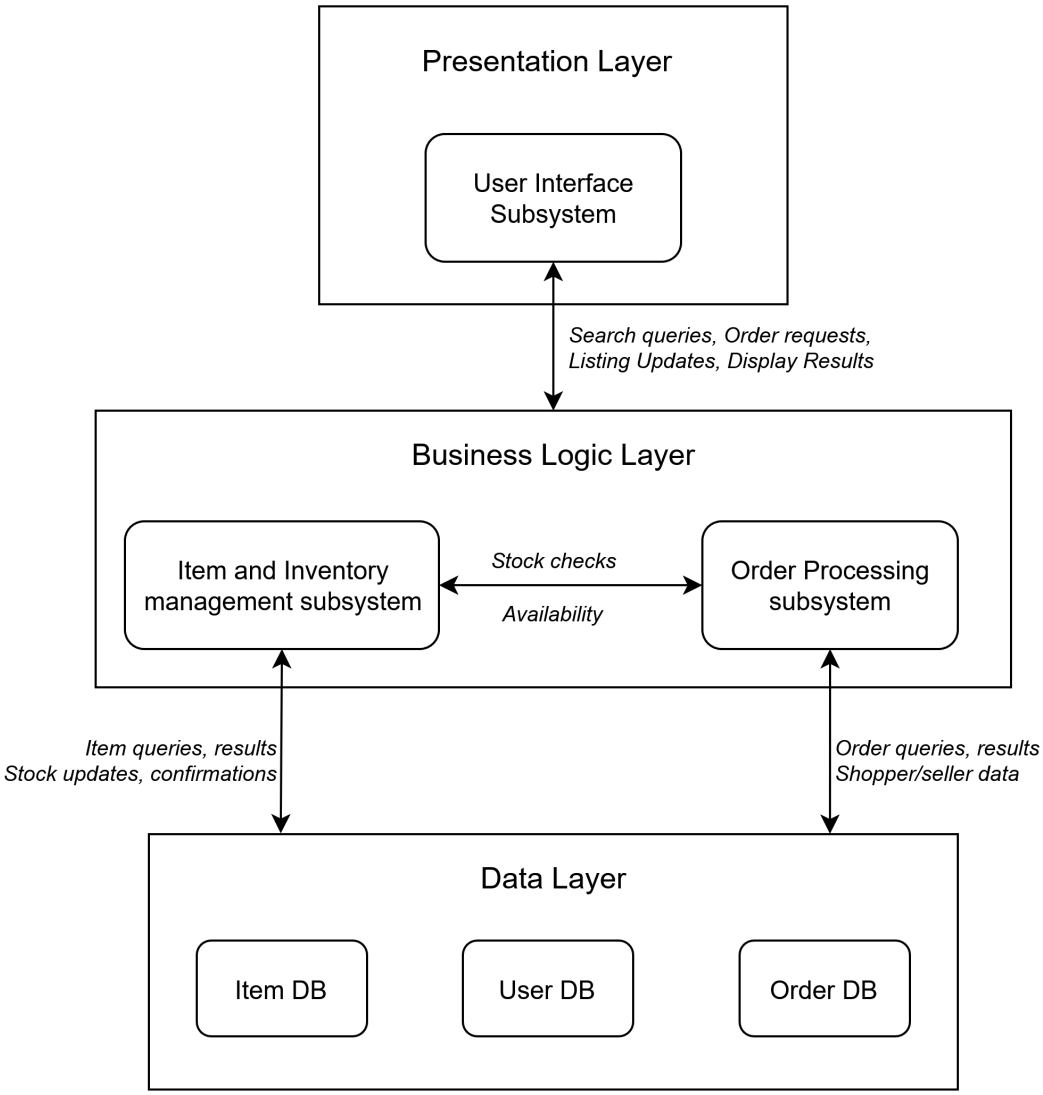

Figure 1: Block diagram showing the three-tier layered architecture with subsystem interactions

The block diagram (Figure 1) illustrates the three-tier architecture with:

- Presentation Layer containing the User Interface Subsystem
- Business Logic Layer with Item & Inventory Management and Order Processing subsystems
- Data Layer with Item Database, User Database, and Order Database
- Bidirectional arrows showing request-response patterns between layers

2 Part B: Design Pattern Exploration

2.1 Controller Pattern

The **Controller Pattern** is a GRASP (General Responsibility Assignment Software Patterns) principle that assigns the job of receiving and coordinating system operations to a specific class. This class acts as a middleman between the user interface and the business logic. The controller is the first stop after the UI layer, and it delegates work to the right objects instead of doing the business logic itself.

There are two types of controllers: **Use Case Controllers** handle one specific workflow (like ProcessOrderController), while **Façade Controllers** handle all operations for a subsystem (like ShoppingController). The pattern keeps the UI and domain model loosely connected, making it easier to change the interface without breaking business logic. By putting all system event handling in one place, controllers prevent duplicate control logic across multiple UI components.

The controller doesn't have business rules—it just coordinates the objects that do. This makes testing easier since you can test business logic without needing the UI. Controllers are also a good place to add logging, security checks, and transaction management before passing work to domain objects.

Real-World Example: In a **Hospital Patient Management System**, a `PatientAdmissionController` receives requests from the receptionist's interface when admitting a new patient. The controller coordinates everything: it validates patient info with a `PatientValidator`, checks bed availability with a `BedManager`, creates insurance records through an `InsuranceHandler`, and generates documents via a `DocumentGenerator`. The controller doesn't do these tasks itself—it just makes sure they happen in the right order while keeping the UI separate from the backend objects.

2.2 Expert Pattern

The **Information Expert Pattern** (or just Expert Pattern) is another GRASP principle that assigns responsibility to the class that has the information needed to do the job. It keeps related data and behavior together, which is a key part of object-oriented programming. The pattern answers "Which class should handle this task?" with "The class that already has the data to do it."

When we assign responsibilities to information experts, we get classes where data and the methods that use that data live in the same place. This reduces coupling because other classes don't have to pull out information and do calculations themselves—they just ask the expert object to handle it. The code is easier to maintain because changes to how something is calculated only affect the one expert class.

You see this pattern everywhere in OOP. If we need to calculate an order's total price, we ask the Order object (which has the line items) instead of having some other class pull out the line items and do the math. This way, if the calculation changes (like adding discounts or taxes), we only change the Order class.

Real-World Example: In a **Smart Home Energy Management System**, the `EnergyMeter` class calculates total energy consumption for billing. This class has access to hourly readings, peak/off-peak rates, and billing multipliers. Instead of having the billing system pull all this data out and do calculations, the `EnergyMeter.calculateBillAmount()` method handles it. If the utility company changes their rate structure, only the `EnergyMeter` class needs updating. Everything else just calls `meter.calculateBillAmount()` without needing to know how the calculation works.

3 Part C: UML Diagramming

3.1 Use Case Diagram

The use case diagram shows the functional requirements of the Online Shopping Platform from the users' perspectives.

Actors:

- Shopper (primary user who buys items)
- Seller (manages item listings and inventory)
- Payment System (external system)

Use Cases:

- Browse Items
- Search Items
- Place Order
- Process Payment
- View Order History
- Manage Listings

Key Relationships:

- "Place Order" *includes* "Process Payment" (placing an order always includes payment processing)
- "Search Items" *extends* "Browse Items" (search is optional enhancement of browsing)

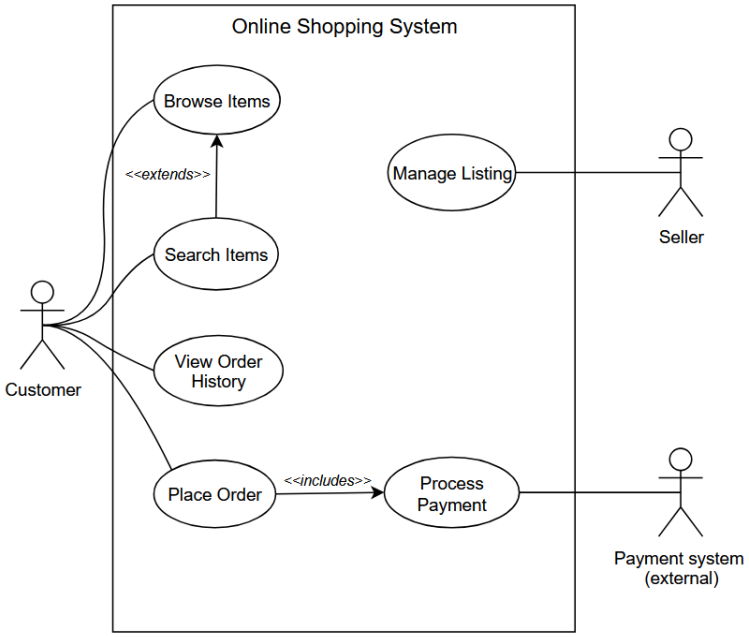

Figure 2: Use Case Diagram showing actors and their interactions with the system

3.2 Class Diagram (Applying Controller and Expert Patterns)

The class diagram demonstrates the application of Controller and Expert patterns in the system design.

Key Classes:

- **ShoppingController** (Controller Pattern) - Coordinates system operations
- **ShoppingCart** (Expert Pattern) - Knows cart contents and calculates totals
- **Order** (Expert Pattern) - Knows order details and calculates subtotals, tax, and final totals
- **Item** (Expert Pattern) - Knows its own price, availability, and stock quantity
- **ItemManager** - Manages item catalog and inventory
- **OrderProcessor** - Creates orders and processes payments
- **CartItem** - Represents an item in the shopping cart (part of ShoppingCart)
- **OrderItem** - Represents an item in an order (part of Order)

Relationships (following textbook notation):

- ShoppingController uses ItemManager, OrderProcessor, and ShoppingCart (dependencies)
- ItemManager manages Items (dependency)
- ShoppingCart contains CartItems (composition ⧫, 1..*)
- Order contains OrderItems (composition ⧫, 1..*)
- CartItem references exactly 1 Item (association)
- OrderItem references exactly 1 Item (association)
- OrderProcessor creates Orders (dependency)

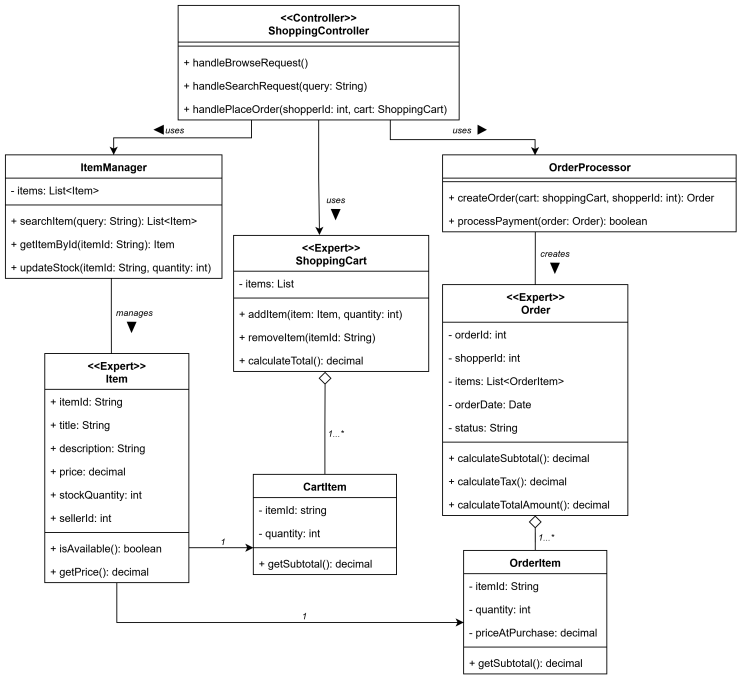

Figure 3: Class Diagram showing Controller and Expert pattern applications

3.3 Activity Diagram (Core Workflow: Place Order)

The activity diagram illustrates the complete workflow for placing an order, organized into three swimlanes representing different actors and systems.

Swimlanes:

1. **Shopper** - Customer actions (browsing, selecting items, confirming order)
2. **System** - Automated shopping platform processes (checking availability, validating credentials, creating orders)
3. **Payment System** - External payment processing

Key Workflow Steps:

- Browse/search items and select item
- System checks item availability before adding to cart
- Shopping loop allows adding multiple items
- Proceed to checkout requires login/authentication
- Review order summary with option to modify cart
- System verifies item availability and reserves inventory (parallel activities)
- Payment system processes transaction with retry/cancel options
- System creates order, updates inventory, and sends notifications (parallel to shopper and seller)
- Shopper views order confirmation

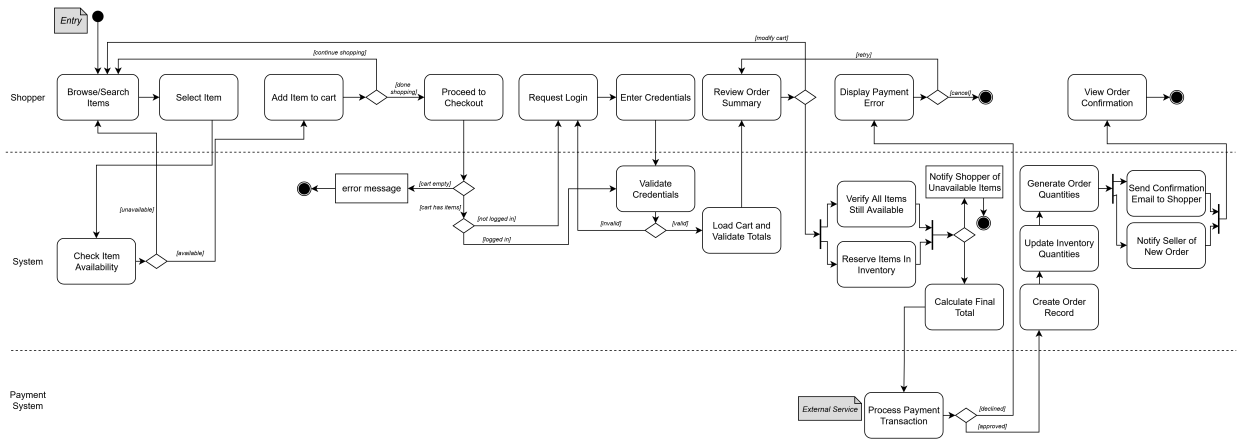

Figure 4: Activity Diagram showing the complete place order workflow with swimlanes

4 Part D: Pattern Application Reflection

4.1 How Patterns Improve System Design

Using the **Controller** and **Expert** patterns makes the Online Shopping Platform design much better in several ways. These patterns give us proven solutions to common problems and make the system easier to maintain and scale.

4.2 Problems Each Pattern Solves

4.2.1 Controller Pattern

The **Controller Pattern** fixes the problem of **tight coupling between the UI and business logic**. Without a controller, UI components would directly create and manipulate domain objects, which causes several issues:

1. **Scattered system logic**: Every UI component (web page, mobile screen, API endpoint) would need to know the exact sequence of operations for complex workflows. In our shopping platform, placing an order means checking inventory, validating the cart, processing payment, and updating multiple subsystems. Without a controller, this coordination logic gets duplicated everywhere.

2. **Hard to change the UI**: If we want to add a mobile app next to our web interface, we'd have to rewrite all the coordination logic. The Controller Pattern fixes this by putting all system operations in one place.

3. **Hard to test**: Testing business logic is tough when it's mixed with UI code. With a controller, we can test the order placement workflow without needing any UI at all.

In the shopping platform, the `ShoppingController` acts as a single entry point. When a shopper places an order, the UI just calls `controller.handlePlaceOrder()`. The controller then talks to `ItemManager` to check stock and `OrderProcessor` to create the order. If we need to add something new (like fraud detection), we only change the controller, not every UI component.

4.2.2 Expert Pattern

The **Expert Pattern** solves the problem of **deciding which class should do what**. It answers "Where should this method go?" and prevents several bad practices:

1. **Feature envy**: Without Expert Pattern, external classes would constantly pull data out of objects and do calculations themselves. For example, a billing service might extract all items from a cart, loop through them, grab each book's price, and calculate the total. This breaks encapsulation and spreads calculation knowledge all over the code.

2. **Weak cohesion**: When methods end up in classes that don't have the data they need, those classes get messy and confusing. Expert Pattern keeps things focused by putting related data and behavior together.

3. **Fragile design**: When calculation logic lives outside the data classes, changing the data structure means updating code in multiple places.

In the shopping platform, the `ShoppingCart` class shows Expert Pattern in action. It has `calculateTotal()` because it's the expert—it knows what items are in the cart. The `Order` class calculates its own subtotal, tax, and total because it has the order items. The `Item` class has `isAvailable()` because it knows its own stock quantity. If we change how prices work (adding discounts, membership pricing, etc.), we only change the expert class that handles that calculation.

4.3 Benefits for Large Teams and Evolving Codebases

These patterns become way more valuable as projects get bigger and more complex. Here's how they help:

4.3.1 For Large Teams

1. **Clear ownership**: Controller Pattern sets clear boundaries between frontend and backend developers. The frontend team just needs to call controller methods, while the backend team handles coordination logic. Expert Pattern makes it obvious who owns what—the order processing team owns the Order class and its calculations.

2. **Parallel development**: Different people can work on different parts without stepping on each other's toes. One dev can improve the `InventoryManager` while another works on `OrderProcessor`, as long as they respect the controller's interface. Expert Pattern helps by keeping functionality in one place, reducing merge conflicts.

3. **Easier code reviews**: Reviewers can quickly spot when responsibilities are wrong. If someone puts order total calculations in the `ShoppingController` instead of the `Order` class, it's obviously breaking Expert Pattern. Having this shared vocabulary helps teams talk about design.

4. **Easier onboarding**: New team members understand the system faster when patterns are used consistently. They learn that controllers coordinate things and experts handle their own data, making the code more predictable.

4.3.2 For Evolving Codebases

1. **Less impact from changes**: When requirements change, patterns limit the damage. Adding a new payment method? Just modify `OrderProcessor`, and the controller calls the same method. UI stays the same. Discount rules change? Update the `Order` class's calculation methods, not every place that shows prices.

2. **Easier refactoring**: Patterns give you stable structures to work with. If inventory checking is slow, we can optimize `ItemManager` without touching the controller or UI. The separation means we can even swap out entire components as long as the interface stays the same.

3. **Technology migration**: When tech changes, we might go from web to mobile or add a REST API. Controller Pattern makes this doable because business logic isn't tied to any specific UI. We just create new UI components that call the same controller methods.

4. **Better testing**: Both patterns make testing easier. Controllers can be tested with mocks for their dependencies. Expert objects can be unit tested alone—we can test `Order.calculateTotal()` without needing UI, databases, or anything else. This makes automated testing way easier.

5. **Self-documenting code**: When patterns are used consistently, code explains itself. A dev looking at `ShoppingCart.calculateTotal()` immediately knows this is where cart calculations happen. No searching through multiple files wondering where totals get computed.

4.4 Example: Adding a Membership Discount Program

Here's a practical example showing why these patterns matter. Say the business wants tiered discounts for members:

Without patterns:
- Hunt through the entire codebase for every price calculation
- UI components probably have hardcoded math
- Different screens might calculate discounts differently
- Testing means running the whole app

With patterns:
- **Expert Pattern**: Add `calculateDiscount()` to the `Order` class and update `calculateTotal()` to use it
- **Controller Pattern**: Controller keeps calling `order.calculateTotal()`—doesn't need to know about discounts
- UI just displays whatever total the order returns—no changes needed
- Write unit tests for `Order.calculateDiscount()` by itself
- The change stays in one place, less risk and faster development

This separation is what lets systems evolve over years without becoming unmaintainable. Without patterns, codebases turn into "legacy" systems where every change is risky and expensive.

4.5 Conclusion

Controller and Expert patterns turn the Online Shopping Platform from a potential mess of tangled components into a clean, maintainable system. Controller Pattern gives us clear coordination points and keeps UI separate from business logic. Expert Pattern makes sure responsibilities go to the classes that have the right information. Together, they build a foundation that supports team work, handles changing requirements, and stays understandable as the system grows. These patterns come from decades of experience figuring out what makes software maintainable, and using them here shows why they're core principles of good object-oriented design.

4.6 Diagram Reference

For quick navigation, all diagrams included in this assignment are listed below:

- **Figure 1**: Block Diagram (page 5) - Shows the three-tier layered architecture with subsystem interactions
- **Figure 2**: Use Case Diagram (page 8) - Shows actors and their interactions with the system
- **Figure 3**: Class Diagram (page 9) - Demonstrates Controller and Expert pattern applications
- **Figure 4**: Activity Diagram (page 10) - Illustrates the complete place order workflow with swimlanes

---

References

- Kung, David C. (2013). *Software Engineering: An Object-Oriented Perspective* (2nd ed.). Course textbook chapters 3.4, 3.5, 10.1–10.6.