Lab 01: Pattern Recognition & The Rule of Three

Week 1 Assignment

Student Name: _____ Date: _____

Overview

Total Points: 100 (50 points per lab)
Required Reading: *Learning JavaScript Design Patterns, 2nd Ed.* by Addy Osmani
Chapter 1: "Introduction to Design Patterns" (Sections: "What Is a Pattern?", "An Everyday Use Case for Design Patterns")
Chapter 2: "Pattern-ity Testing, Proto-Patterns, and the Rule of Three" (Sections: "The Pattern Tests", "Rule of Three")

Key Vocabulary

Before beginning, ensure you understand these terms from the book:

- **Pattern**: A reusable solution template that can be applied to recurring problems in software design (Ch. 1, "What Is a Pattern?").
- **Proto-Pattern**: A pattern that has not yet conclusively passed the "pattern"-ity tests (Ch. 2, "The Pattern Tests").
- **Rule of Three**: A validation criterion requiring a pattern to demonstrate: (1) Fitness of Purpose, (2) Usefulness, (3) Applicability (Ch. 2, "Rule of Three").
- **Pattern-ity Tests**: Four criteria a good pattern must meet: solves a particular problem, doesn't have an obvious solution, describes a proven concept, describes a relationship (Ch. 2, "The Pattern Tests").
- **Anti-Pattern**: A common response to a problem that incurs more liability than benefit (Ch. 4, referenced).

1. Lab 1.1: The Pattern Audit (50 Points)

Objective

Analyze legacy code to determine if it represents a genuine design pattern or merely an ad-hoc solution. This lab tests your understanding of pattern recognition and the Rule of Three validation criteria.

Background

According to Chapter 2 of the textbook, not every solution qualifies as a pattern. A proto-pattern must pass rigorous validation before being recognized as a pattern. This exercise requires you to apply the academic criteria from the book to evaluate real code.

Provided Code

You are given the following JavaScript code snippet from a legacy e-commerce application:

Listing 1: Legacy Shopping Cart Implementation

```
// shoppingCart.js - Legacy implementation
var cart = [];
var total = 0;

function addItem(name, price) {
    cart.push({ name: name, price: price });
    total += price;
    updateDisplay();
}

function removeItem(index) {
    var item = cart[index];
    total -= item.price;
    cart.splice(index, 1);
    updateDisplay();
}

function updateDisplay() {
    var cartDiv = document.getElementById('cart');
    cartDiv.innerHTML = '';
    cart.forEach(function(item, index) {
        var itemDiv = document.createElement('div');
        itemDiv.innerHTML = item.name + ' - $' + item.price;
        var removeBtn = document.createElement('button');
        removeBtn.onclick = function() { removeItem(index); };
        itemDiv.appendChild(removeBtn);
        cartDiv.appendChild(itemDiv);
    });
    document.getElementById('total').textContent = '$' + total;
}

function calculateTax(rate) {
    return total * rate;
}

function checkout() {
    var tax = calculateTax(0.08);
    var finalTotal = total + tax;
```

```
    alert('Total: $' + finalTotal);
    cart = [];
    total = 0;
    updateDisplay();
}
```

Tasks

Task 1: Pattern Identification (15 points)

1. Identify any design patterns (or proto-patterns) you observe in this code.
2. For each identified pattern, state which category it belongs to (Creational, Structural, or Behavioral) based on Chapter 6 of the textbook.
3. Explain why you believe it is or is not a genuine pattern.

Task 2: Rule of Three Evaluation (20 points) Apply the Rule of Three criteria from Chapter 2, Section "Rule of Three" to evaluate if this code represents a pattern:

1. **Fitness of Purpose (7 points)**: Does this solution successfully solve a particular problem? What problem does it solve? Provide evidence from the code.
2. **Usefulness (7 points)**: Does this solution provide a proven benefit? What benefits does it offer? Are there limitations?
3. **Applicability (6 points)**: Is this solution applicable in a broad range of contexts, or is it specific to this one scenario? Explain your reasoning.

Task 3: Pattern-ity Tests (10 points) Evaluate the code against the four pattern-ity tests from Chapter 2, Section "The Pattern Tests":

1. Does it solve a particular problem? (2.5 points)
2. Does it have an obvious solution? (2.5 points)
3. Does it describe a proven concept? (2.5 points)
4. Does it describe a relationship between system structures? (2.5 points)

Task 4: Anti-Pattern Identification (5 points) Identify at least one anti-pattern present in this code. Reference Chapter 4 concepts (global namespace pollution, tight coupling, etc.). Explain why it qualifies as an anti-pattern and suggest a solution.

Deliverable

Submit a written report (PDF or Word document) containing:

- Your analysis for each task above
- Code references (line numbers) supporting your conclusions
- References to specific book sections (e.g., "Ch. 2, 'Rule of Three'")
- A conclusion stating whether this code represents a pattern, proto-pattern, or neither

Lab 1.1 Assessment Rubric (50 points)

Task-Specific Criteria:

- **Task 1 - Pattern Identification (15 pts)**:
  - Correctly identifies patterns/proto-patterns: 8 pts
  - Correctly categorizes patterns (Creational/Structural/Behavioral): 4 pts
  - Provides clear reasoning with code references: 3 pts
- **Task 2 - Rule of Three (20 pts)**:
  - Fitness of Purpose analysis with evidence: 7 pts
  - Usefulness analysis (benefits & limitations): 7 pts
  - Applicability analysis (context range): 6 pts
- **Task 3 - Pattern-ity Tests (10 pts)**:
  - Solves particular problem: 2.5 pts
  - No obvious solution: 2.5 pts
  - Proven concept: 2.5 pts
  - Describes relationship: 2.5 pts
- **Task 4 - Anti-Pattern Identification (5 pts)**:
  - Correctly identifies anti-pattern with Chapter 4 reference: 3 pts
  - Provides viable solution: 2 pts

Grading Notes: Each criterion is assessed as: **Excellent (100%)**, **Good (75%)**, **Satisfactory (50%)**, or **Insufficient (0%)**

2. Lab 1.2: Provider Pattern Implementation (50 Points)

Objective

Implement the Provider Pattern example from Chapter 1, Section "An Everyday Use Case for Design Patterns" to solve the prop drilling problem in a React application.

Background

The textbook (Ch. 1, "An Everyday Use Case for Design Patterns") demonstrates how the Provider Pattern solves the problem of component trees needing access to shared data. Traditional prop drilling becomes unmaintainable as component hierarchies deepen. This lab requires you to implement this pattern using React's Context API.

Problem Statement

You are building a user dashboard application. Multiple nested components need access to:

- User information (name, email, role)
- User permissions (canEdit, canDelete, canView)
- Theme preferences (darkMode, fontSize)

Without the Provider Pattern, you would need to pass these props through every level: `App → Dashboard → Header → UserMenu → UserInfo`.

Tasks

Task 1: Context Creation (15 points)

1. Create a `UserContext.js` file using `React.createContext()`.
2. Define the context structure to hold: user info, permissions, and theme preferences.
3. Export both the Context object and a custom hook `useUser()` that wraps `useContext(UserContext)`.
4. Include proper error handling if the hook is used outside the Provider.

Task 2: Provider Implementation (20 points)

1. Create a `UserProvider.js` component that:
   - Accepts `children` as props
   - Manages user state using `useState`
   - Provides initial mock data (hardcoded user object)
   - Wraps children with `UserContext.Provider`
2. The Provider must pass the complete context value (user, permissions, theme) to consumers.
3. Include TypeScript types or PropTypes for type safety (bonus: 3 points).

Task 3: Consumer Implementation (10 points) Create three nested components demonstrating the pattern:

1. `Dashboard.js` - Uses `useUser()` to display user name

2. `Header.js` - Uses `useUser()` to show user role and theme toggle
3. `UserMenu.js` - Uses `useUser()` to display permissions and email

Each component must be at least 2 levels deep in the component tree to demonstrate that prop drilling is eliminated.

Task 4: Documentation (5 points) Write a brief explanation (200-300 words) describing:

- How this implementation follows the Provider Pattern from the textbook
- The problem it solves (prop drilling)
- The benefits over traditional prop passing
- Reference to Chapter 1, Section "An Everyday Use Case for Design Patterns"

Technical Requirements

- Use React 18+ with functional components and hooks
- Code must be properly formatted and commented
- Include error boundaries for context usage
- Submit working code that compiles and runs without errors

Deliverable

Submit:

- All source code files (`UserContext.js`, `UserProvider.js`, component files)
- Screenshot of the running application showing all three components displaying data
- Documentation file (Task 4)
- A brief README explaining how to run the project

Lab 1.2 Assessment Rubric (50 points)

Task-Specific Criteria:

- **Task 1 - Context Creation (15 pts)**:
  - Correct `createContext` usage: 5 pts
  - Custom hook `useUser()` implemented: 5 pts
  - Error handling (hook outside Provider): 5 pts
- **Task 2 - Provider Implementation (20 pts)**:
  - Provider component structure (children, state): 8 pts
  - State management (useState, mock data): 6 pts
  - Context value passing (complete data): 6 pts
  - TypeScript/PropTypes (bonus): +3 pts
- **Task 3 - Consumer Implementation (10 pts)**:
  - Three components created (Dashboard, Header, UserMenu): 6 pts
  - Correct `useUser()` hook usage in each: 4 pts
- **Task 4 - Documentation (5 pts)**:
  - Clear explanation (200-300 words): 3 pts
  - Book reference (Ch. 1, Section): 2 pts

Grading Notes: Code must compile and run. Non-functional code receives 0 points for Task 2 & 3.

Submission Requirements and Regulations

OquLabs Protocol (30 points)

All labs must be completed using OquLabs with the following requirements:

- **Full Screen Mode (10 points)**: Maintain full screen mode throughout the session. Background applications or partial screen mode will result in a **-20 point penalty**.
- **Active Typing (10 points)**: All code must be typed actively. Copy-paste detection will result in a **-30 point penalty**. Boilerplate templates are allowed, but logic must be typed.
- **Session Duration (10 points)**: Minimum session duration of 30 minutes. Suspiciously fast submissions (< 30 mins) will result in a **-10 point penalty** and may be considered as absence.

Note: OquLabs session logs serve as digital attendance records. Short sessions may be treated as absence.

Git Discipline (10 points)

All code submissions must follow professional Git practices:

- **Folder Structure (5 points)**: Organize code in proper folder structure: `Lab_01/` containing subdirectories (e.g., `Lab_01/task1/`, `Lab_01/task2/`). Flat file dumps or root-level submissions will result in a **-10 point penalty**.
- **Conventional Commits (5 points)**: Use conventional commit messages (e.g., `feat: add UserContext implementation`, `fix: resolve Provider state issue`). Bad commit messages will result in a **-5 point penalty**.
- **Incremental History**: Submit with at least 3 meaningful commits showing incremental progress. Single file dumps will result in a **-5 point penalty**.

Code Quality Standards (20 points)

- **Naming Conventions (10 points)**: Use consistent camelCase naming for variables and functions (e.g., `userContext`, `createUserProvider`). Inconsistent naming will result in a **-5 point penalty**.
- **Comments (10 points)**: Include meaningful comments explaining *why* code exists, not just *what* it does. Zero comments will result in a **-10 point penalty**.

AI Usage Policy

If you use AI tools (ChatGPT, Copilot, etc.) during development:

- **AI Report Required**: You must submit an `AI_REPORT.md` file documenting:
  - Which AI tool was used
  - Specific prompts you used
  - How you modified/verified the AI-generated code
  - What you learned from the process
- **Penalties**: Missing AI report when AI was used: **-100 points** (Academic Dishonesty). Lazy or incomplete reporting: **-20 points**.
- **Transparency**: Using AI is allowed, but attribution is mandatory. This follows academic citation standards.

Submission Instructions

1. **Lab 1.1**: Submit your analysis report as a PDF
2. **Lab 1.2**: Submit your code as a Git repository (GitHub/GitLab link) or ZIP file containing:
   - All source files in proper folder structure (`Lab_01/task1/`, `Lab_01/task2/`)
   - Screenshots of running application
   - Documentation file
   - `AI_REPORT.md` (if AI was used)
   - README explaining how to run the project
3. Include your name, student ID, and date on all submissions

4. Submit through the course portal by the deadline
5. Ensure OquLabs session logs are accessible for verification

Comprehensive Assessment Summary

Total Lab Score Breakdown (100 points per lab)

The final grade for each lab combines task-specific points with general requirements:

Category	Points	Assessment Method
Lab-Specific Tasks		
Lab 1.1: Pattern Analysis Tasks	50	See Lab 1.1 Rubric
Lab 1.2: Implementation Tasks	50	See Lab 1.2 Rubric
Functionality & Code Quality		
Code runs without errors	20	[YES] Runs / [NO] Doesn't run (-20)
Edge cases handled	20	-5 per failed test case
CamelCase naming conventions	10	[YES] Consistent / [NO] Inconsistent (-5)
Comments (Why vs What)	10	[YES] Present / [NO] Zero comments (-10)
OquLabs Protocol		
Full Screen Mode maintained	10	[YES] Full screen / [NO] Partial (-20)
Active Typing (No Copy-Paste)	10	[YES] Typed / [NO] Paste detected (-30)
Session Duration > 30 mins	10	[YES] > 30 min / [NO] < 30 min (-10)
Git Discipline		
Folder Structure (Lab_01/...)	5	[YES] Proper / [NO] Flat dump (-10)
Conventional Commits (feat: fix:)	5	[YES] Proper / [NO] Bad messages (-5)
Incremental History (3+ commits)	0	[YES] 3+ commits / [NO] Single dump (-5)
Documentation		
AI Report (if AI used)	0	[YES] Included / [NO] Missing (-100)
AI Strategy Explained	0	[YES] Detailed / [NO] Lazy (-20)
TOTAL	**100**	

Grading Process:

1. Grade task-specific criteria (50 points) using detailed rubrics
2. Test functionality: Run code, check edge cases (40 points)
3. Review OquLabs logs: Verify protocol compliance (30 points)
4. Check Git repository: Structure, commits, history (10 points)
5. Assess code quality: Naming, comments (20 points)
6. Verify documentation: AI report if applicable
7. Apply penalties for violations
8. Calculate final score (max 100 points per lab)

Academic Integrity

All work must be your own. You may reference the textbook and official React documentation, but code must be written by you. Plagiarism, unauthorized collaboration, or failure to disclose

AI usage will result in a zero grade and academic penalties. OquLabs monitoring ensures fair assessment for all students.