SOFTWARE DESIGN DOCUMENT

System Overview

The **KitHub system** is a web-based application designed to support temporary borrowing of campus equipment such as lab tools, cameras, and sports gear. It aims to enhance sustainability and reduce costs by allowing students to borrow instead of purchase rarely used equipment.

The selected use case for this assignment is **Borrow Equipment**. This is a core operation in the system and involves:

- Students requesting an item
- The system evaluating the request using predefined borrowing strategies
- The administrator approving or rejecting the request

The use case implements the **Strategy Design Pattern** to dynamically switch between borrowing rules based on user type or equipment category.

System Context

The KitHub system operates as a standalone web application designed for use within a university campus. It interacts primarily with two user types: **students** and **administrators**.

This system does not integrate with any external services or university databases in its MVP version, making it lightweight and easy to deploy. The only system boundaries include internal data validation and access restrictions based on roles.

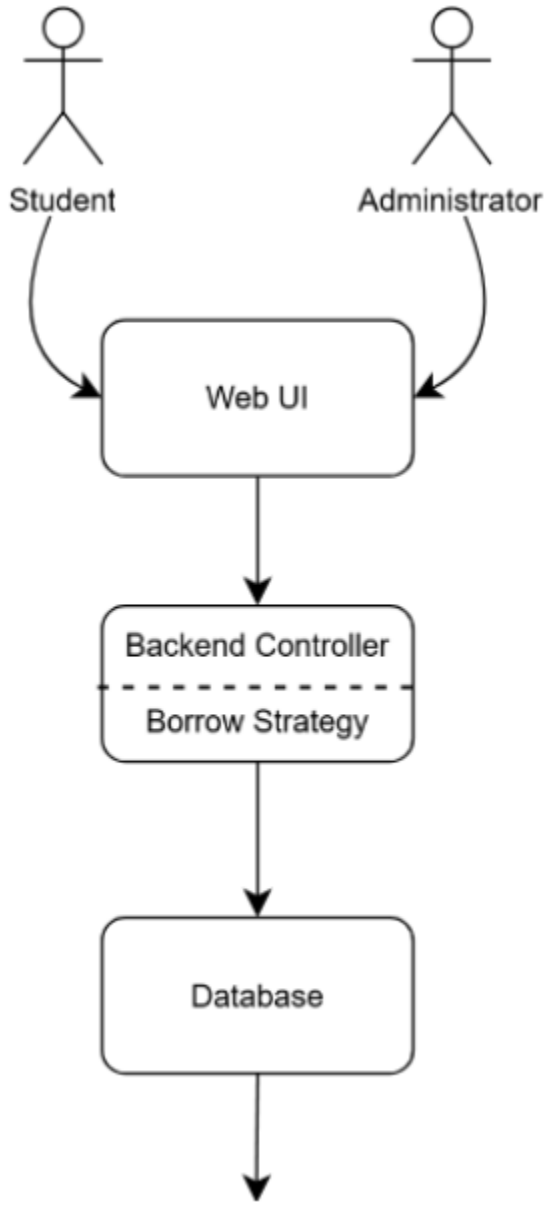
Actors and Interactions

Actor	Interaction Type	Purpose
Student	Web UI / Borrow Module	Requests to borrow available items
Administrator	Web UI / Admin Panel	Approves or denies borrow requests
System	Internal logic (Strategy Pattern)	Selects borrowing strategy based on context (user or item type)
Database	MySQL (internal)	Stores borrow records, users, items

---

Context Diagram (Simplified)

- Users interact via browser (client-side).
- Frontend connects to backend controller.
- Backend logic invokes appropriate borrowing strategy.
- Database stores/updates state of requests and items.


Student
Administrator
Web UI
Backend Controller
Borrow Strategy
Database

Key Features and Functionality

Overview

This implementation of the “Borrow Equipment” use case offers a comprehensive and adaptable system designed to efficiently manage item borrowing requests within a campus setting. It allows logged-in students to browse and select available equipment from a centralized catalog. Once an item is chosen, students can fill out a structured borrow form to specify their preferred borrowing times and include any relevant notes. This ensures a user-friendly and intuitive process for initiating equipment loans.

Flexible Borrowing Rules with Strategy Pattern

A standout feature of the system is its use of the Strategy Pattern to manage borrowing rules. This design approach enables the encapsulation of various borrow validation strategies based on factors such as the user’s role, the type of equipment, and previous borrowing history. The flexibility of this architecture allows administrators to introduce or adjust borrowing policies without having to alter the core borrow request logic, making the system highly maintainable and adaptable to evolving needs.

Admin Approval Workflow

Once a student submits a request, it enters an administrative workflow for approval. Admins can review pending requests from their dashboard, approve or deny them, and include optional notes when providing feedback. Decisions made by the admin automatically trigger notifications to keep students informed of the outcome, enhancing communication and transparency.

Real-Time Status Management

The system also ensures real-time status management of all equipment. When a request is approved or an item is returned, the equipment’s availability is updated instantly. Users can view their current borrow status and history through their personal dashboards, which promotes accountability and convenience.

Modular and Extensible Architecture

Underpinning this functionality is a modular and extensible system architecture. The borrowing logic is broken into separate, reusable components, which not only improves testability but also minimizes code duplication. This setup allows for easy integration of new rules or features—such as giving priority to certain student groups or imposing limits on borrow durations—without disrupting existing functionality.

Input Validation and User Feedback

Finally, the system includes thorough input validation and responsive user feedback. It checks for scheduling conflicts, ensures equipment is available, and confirms compliance with applicable rules before processing any requests. Users receive clear, contextual error or confirmation messages, helping them understand the status of their request and guiding them toward successful submissions.

Assumptions and Dependencies

Assumptions

- **User Authentication Is Required**
  It is assumed that users must be authenticated (i.e., logged in) before they can initiate a borrow request.
- **Only Available Items Can Be Borrowed**
  The borrow operation only applies to items marked as available. Unavailable or already borrowed items are filtered out automatically.
- **Borrow Strategy Is Selected Based on Item Type or Policy**
  The appropriate borrow rule strategy (e.g., MaxDurationStrategy, RestrictedAccessStrategy) is selected at runtime based on the item's category or institutional rules.
- **Admin Approval Is Always Required**
  Every borrow request goes through a manual approval by an administrator. The system does not automatically grant borrow permissions.
- **Single Borrow per User per Item**
  A user cannot borrow multiple instances of the same item simultaneously.
- **Time Slots and Conflicts Are Pre-validated**
  Borrow time inputs from users are assumed to be validated against the existing booking records.

Dependencies

- **Authentication Service**
  Used to verify user identity and role before processing borrow logic.
- **Inventory Management Module**
  Supplies item availability data and updates item status post-approval or rejection.
- **Notification System**
  Sends status updates to both students and admins regarding the borrow request.
- **Borrow Strategy Interface and Implementations**
  A set of strategy classes define the rules for different borrowing policies (e.g., max borrow length, eligibility, item restrictions).

Architectural Design

1. System Architecture Diagram (High-Level)

The architecture follows a **Layered (n-tier)** approach:

- **Presentation Layer**: Reacts to user input, displays borrowing form, and renders feedback.
- **Application Layer**: Contains the borrow controller and the strategy handler. It orchestrates the logic.
- **Domain Layer**: Hosts the strategy pattern interface and its concrete implementations (e.g., StandardBorrowStrategy, LabOnlyBorrowStrategy).
- **Data Access Layer**: Communicates with the database to store and retrieve item and borrow request information.
- **Database Layer**: MySQL tables for Users, Items, BorrowRequests.

---

2. Architectural Style

Pattern: Layered Architecture

Justification:

- Clean separation of concerns.
- Aligns with modular design principle, useful when applying design patterns like Strategy.

---

3. Strategy Pattern Integration

In the application layer, a BorrowHandler class is responsible for selecting the right borrowing rule at runtime:

```
# Pseudocode
 strategy = get_strategy_for_user_and_item(user, item)
 if strategy.is_eligible(user, item):
     proceed_with_borrow()
 else:
         reject_borrow()
```

The Strategy interface supports pluggable policies such as:

- **StandardBorrowStrategy**: Allows regular users to borrow standard items with default rules.
- **RestrictedBorrowStrategy**: Applies tighter rules, for example, limiting access to high-value or sensitive items.
- **MaxDurationBorrowStrategy**: Enforces a maximum borrowing duration depending on user or item type.

Component Design

Subsystems and Modules

The “Borrow Equipment” system is structured into well-defined subsystems and modules, each responsible for a specific aspect of the borrowing process. At the core of the architecture is the BorrowController, which acts as the entry point for handling borrow requests originating from the frontend. It delegates the processing logic to the BorrowService, which contains the primary business logic and coordinates interactions with other components.

To enforce borrowing rules, the system utilizes a flexible strategy design pattern. This is implemented via the BorrowStrategy interface, which defines the core contract for all strategy classes. Concrete implementations such as StandardBorrowStrategy and RestrictedBorrowStrategy apply specific eligibility rules depending on user roles, item categories, or historical borrowing data. The persistence of borrow requests is managed by the BorrowRepository, which communicates with a MySQL database. Meanwhile, the NotificationService is responsible for alerting users and administrators about the status of their requests through emails or dashboard notifications.

Component Responsibilities

The BorrowController exposes a borrow request endpoint (e.g., /api/borrow) and acts as a conduit between the frontend and backend systems. Upon receiving a request, it packages the user and item data and forwards it to the BorrowService.

The BorrowService orchestrates the business logic by selecting the appropriate borrowing strategy through a StrategySelector, validating the request, and deciding whether to persist it or return an error. It serves as the central hub connecting validation, data management, and user communication.

All borrowing rules are encapsulated within implementations of the BorrowStrategy interface. Each strategy must define two core methods: is_eligible(user, item) to assess whether the borrow request meets defined criteria, and get_max_duration() to enforce duration limits.

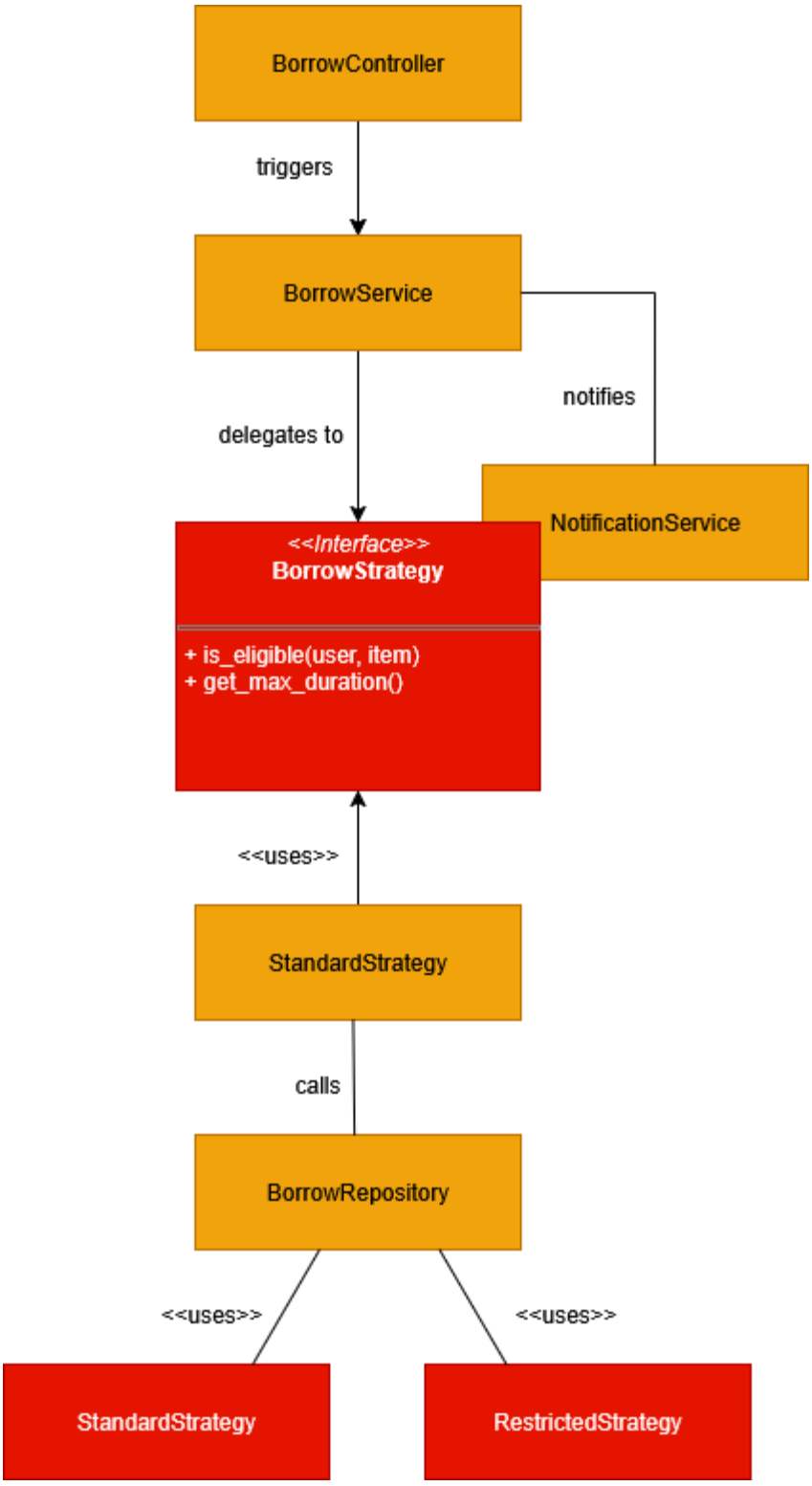
Data operations are handled by the BorrowRepository, which executes the SQL queries necessary to save requests and retrieve their status. On the communication front, the NotificationService generates and delivers context-aware messages, informing users and admins of approval or rejection outcomes.

Interfaces Between Components

The interaction flow between components is both logical and decoupled. The BorrowController communicates directly with the BorrowService, which in turn interacts with multiple subsystems:

the selected BorrowStrategy for rule validation, the BorrowRepository for database operations, and the NotificationService for user updates.

This modular setup ensures high cohesion within each component and low coupling between components. As a result, the system remains maintainable, extensible, and testable—ideal for scaling or integrating new features in the future.

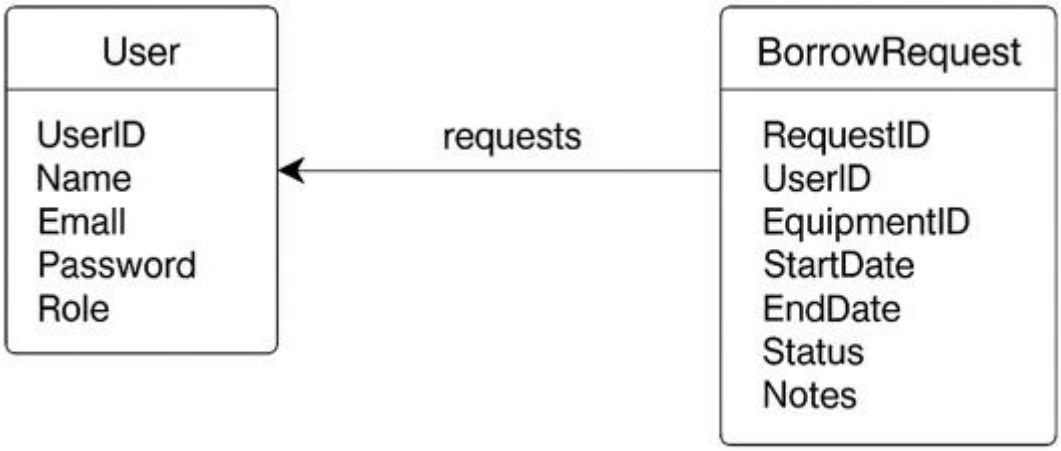

BorrowController
triggers
BorrowService
notifies
delegates to
NotificationService
<<Interface>>
BorrowStrategy
+ is_eligible(user, item)
+ get_max_duration()
<<uses>>
StandardStrategy
calls
BorrowRepository
<<uses>>
<<uses>>
StandardStrategy
RestrictedStrategy

Data Design

Data Model / ER Diagram

The system uses a relational database to manage users, equipment, and borrow transactions. The key entities involved in the **“Borrow Equipment”** use case are:

- **User** (UserID, Name, Email, Password, Role)
- **Equipment** (EquipmentID, Name, Category, Status)
- **BorrowRequest** (RequestID, UserID, EquipmentID, StartDate, EndDate, Status, Notes)



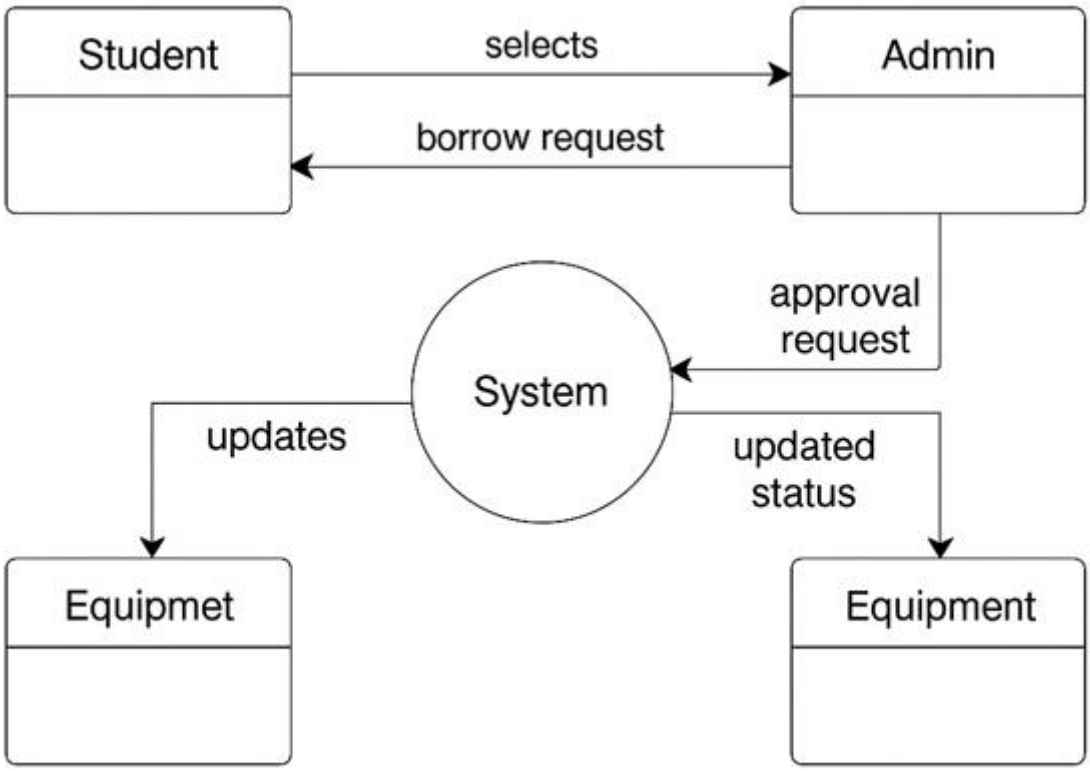
Data Storage (Database or File Structure)

The data is stored in a **MySQL database**. Table definitions:

- **Users Table**: Stores user credentials and roles (student/admin).
- **Equipment Table**: Stores equipment metadata and availability status.
- **BorrowRequests Table**: Tracks each borrow request including time range and approval status.

Data Flow Diagrams (DFDs)

Level 1 DFD – Borrow Equipment Use Case



Data Validation Rules

- User input validation for email format, date ranges, and required fields.
- Equipment availability is checked before request is submitted.
- Admin approval must change the borrow request's status from **"pending"** to **"approved"** or **"rejected"**.

Design Patterns

Context

This document outlines the architectural design decisions made in KitHub, focusing on how specific software design patterns were used to improve modularity, scalability, and long-term maintainability.

At its core, KitHub manages the borrowing of campus equipment by users with varying roles—such as students, club members, and administrators—and enforces borrowing rules based on equipment type and user status. To handle this complexity, the Strategy Pattern was applied to encapsulate borrowing logic in role-specific, easily extendable policies.

To support a flexible and secure authentication system, the Factory Method Pattern was used during user registration and login. This approach enables role-based instantiation of user objects (e.g., `StudentUser`, `AdminUser`) and promotes clean separation of concerns in the authentication layer.

For the search and filtering functionality on the equipment catalog, the Decorator Pattern was adopted. It allows users to dynamically apply multiple filters—such as by keyword, category, or availability—without altering the base query logic, enabling composable and scalable UI interactions.

Finally, the Observer Pattern was implemented within the admin module to decouple inventory changes from their side effects, such as logging and notifications. This event-driven architecture improves extensibility and allows the system to respond to state changes in a modular fashion.

Together, these patterns form the architectural foundation of KitHub, enabling robust business logic, a clean separation of responsibilities, and a maintainable codebase that supports future growth.

Why Strategy Pattern?

The application of the Strategy Pattern here aligns closely with solid design principles, particularly the Open/Closed Principle. This means the borrow logic is open for extension but closed for modification. Developers can introduce new borrow policies by simply creating new strategy classes, without touching the core logic already in place. This clear separation of logic also enhances system reliability; since each borrowing strategy is an isolated component, its rules and behavior can be confirmed without impacting other parts of the system. More broadly, the use of this pattern supports a clean separation of concerns, as each strategy class focuses solely on its specific borrowing rules without being entangled in other responsibilities.

How Strategy Pattern’s Applied

The foundation of this approach is a shared interface: BorrowStrategy. This defines a single method, execute_borrow(user, equipment), which every strategy implementation must override. For example, StandardBorrowStrategy provides the default behavior by assigning a 7-day loan period. In contrast, ClubBorrowStrategy extends this privilege, allowing club members to borrow items for up to 14 days. These classes remain focused and concise, each returning a LoanRecord object that captures the terms of the borrow transaction.

To execute borrowing behavior, a BorrowContext class is used. This context is initialized with a specific strategy and exposes a borrow() method. Internally, it simply delegates the borrow logic to the provided strategy. This structure keeps the context flexible and reusable for any future strategies that might be introduced.

Why Factory Pattern?

To support clean, role-based instantiation of user objects, the Factory Method Pattern was implemented in the user registration and login system. Rather than manually assigning roles or handling conditional logic in the controller, the application delegates this to a centralized factory: UserFactory. This approach adheres to the Single Responsibility Principle, keeping the authentication logic modular, scalable, and easier to test.

Future user types (e.g., ClubMemberUser, GuestUser) can be added simply by extending the base User class and updating the factory, without modifying the controller or registration form logic.

How Factory Pattern's Applied

The pattern is implemented in user_factory.py, where a static method create_user(role, name, password) constructs and returns an instance of either StudentUser or AdminUser. These classes inherit from a polymorphic User base model defined using SQLAlchemy.

The auth_controller.py handles routing for /register, /login, and /logout. When a user registers, the factory is invoked to create the appropriate user subclass based on form input. Flask-Login is used to manage session state, and passwords are securely hashed using Werkzeug's security module.

This clean separation ensures that login/register logic remains flexible and role-aware without becoming entangled with the borrow logic.

Why Decorator Pattern?

The Decorator Pattern was applied to support dynamic, layered filtering in the equipment browsing system. It provides a scalable way to apply multiple filters without cluttering the core logic with nested conditionals. By encapsulating each filter in its own decorator class, the system adheres to the Open/Closed Principle—filters can be extended or modified without changing the base query logic. This modularity enables clean, testable, and reusable components, particularly in a UI-driven context where filtering behavior may evolve.

How Decorator Pattern's Applied

The base of this structure is the EquipmentSearch interface, which defines a standard search() method. The BaseEquipmentSearch class implements this interface and performs a basic query for all equipment.

Decorators such as SearchFilterDecorator, CategoryFilterDecorator, and AvailabilityFilterDecorator also implement the EquipmentSearch interface and wrap another EquipmentSearch instance. Each one adds a specific filter condition.

At runtime, the /catalog route dynamically wraps decorators around the base class based on user input (e.g., name, category, availability). The final result is a layered composition of filters applied sequentially and cleanly, ensuring scalability and flexibility.

Why Observer Pattern?

The Observer Pattern was introduced to decouple inventory state management from side-effect operations such as logging and notifications. In the context of the admin module, inventory actions like adding or updating equipment must be able to trigger auxiliary tasks—such as logging the change or sending notifications—without embedding that logic directly into the inventory manipulation code.

This design enables a clean separation of concerns: the `InventoryManager` focuses solely on data manipulation, while observers independently handle post-action responses. This approach greatly improves maintainability and testability, allowing each observer to evolve or scale independently of the core business logic.

How Observer Pattern's Applied

The Observer Pattern is implemented through two abstract base classes defined in `observer.py`: `Subject` and `Observer`. The `InventoryManager` class, located in `inventory_manager.py`, extends `Subject` and is responsible for managing inventory operations.

Upon completing any state-changing action, such as adding new equipment, the `InventoryManager` invokes its `notify()` method, which triggers all registered observers. Observers such as `LoggingObserver` and `NotificationObserver` implement the `Observer` interface and respond to these updates through their `update()` methods.

These observers are registered in `admin_controller.py`, where the `InventoryManager` is instantiated and linked with each observer. This results in a modular, extensible architecture where any new observer behavior (e.g., audit logging, external API sync) can be added without modifying core logic. This pattern also complements Flask's Blueprint structure, keeping admin-specific logic fully self-contained.

Project Structure and Usage

The implementation is cleanly organized under the /src/borrow/ and /src/auth/ directories. Borrowing logic is encapsulated in the borrow_strategy.py and borrow_context.py files, keeping the feature modular and easy to maintain.

User authentication and role-based instantiation are handled in the /src/auth/ module. The login and registration routes are defined in auth_controller.py, and user objects are instantiated via the UserFactory class using the Factory Method Pattern. This separation ensures that authentication logic remains independent of borrowing logic, enhancing modularity and scalability.

In practice, the system selects the appropriate borrow strategy based on the user's role. For example, if the user is a club member, the ClubBorrowStrategy is applied; otherwise, the system defaults to StandardBorrowStrategy. Once the strategy is chosen, the BorrowContext takes over and executes the logic, returning the configured loan details.

Additionally, the browsing and search feature under /catalog uses the Decorator Pattern to apply layered filters dynamically. All filter logic is encapsulated in individual classes within equipment_search.py, located in the src/borrow/ module.

Measurable Improvement

As a result of the refactoring, the system architecture has become more modular, readable, and maintainable. Each component has been separated according to the single-responsibility principle, making the development process easier to manage and extend.

The login and registration features were integrated into the system using Flask Blueprints and the Factory Method Pattern, without modifying the existing controller logic. This approach has clearly demonstrated the extensibility of the system and the benefits of a layered architecture.

In addition, the equipment search and filtering functionality was enhanced using the Decorator Pattern, allowing users to dynamically apply multiple filters. Since each filter is defined in its own class, new filters can be added without altering the core logic.

Most importantly, adding a new borrowing rule, user role, or filter condition no longer requires changes to the existing flow. Developers can simply implement a new class that adheres to the appropriate interface and integrate it into the system. This structure provides a significant improvement in maintainability, scalability, and long-term sustainability.

Implementation Notes

- The “Borrow Equipment” use case was implemented using modular, testable Python components.
- The **Strategy Design Pattern** was applied to abstract borrow rules.
- The **Factory Method Pattern** was applied to generate user instances based on role during registration.
- The **Decorator Pattern** was applied to implement flexible and composable search filters.
- The **Observer Pattern** was applied to decouple inventory management from logging and notification responsibilities within the admin module.
- Authentication was built using Flask-Login, integrated in /src/auth/.
- All logic is structured following a layered architecture and is traceable via commit history.

File Breakdown

File Name	Description
`borrow_strategy.py`	Contains `BorrowStrategy` base class and all specific strategy implementations.
`borrow_context.py`	Context class that accepts a strategy and executes borrowing logic.
`borrow_controller.py`	Handles requests from the UI, chooses appropriate strategy, and creates borrow records.
`models.py`	Defines User, Equipment, and LoanRecord data models.
`auth_controller.py`	Handles login, registration, and logout routes using Flask Blueprints.
`user_factory.py`	Implements Factory Method Pattern to create user objects by role.
`equipment_search.py`	Contains base and decorator classes implementing the Decorator Pattern for search filters.
`observer.py`	Defines abstract base classes for `Subject` and `Observer` used in the Observer Pattern.
`inventory_manager.py`	Implements inventory logic and notifies observers when equipment state changes.
`admin_controller.py`	Blueprint controller for admin routes, including dashboard, inventory, and offer approvals.

Key Technologies Used

- **Backend**: Python 3.x
- **Pattern Framework**: Pure Python, object-oriented implementation
- **Version Control**: GitHub commits traceable to contributors

- **Coding Standards**: PEP8-compliant with docstrings for all public classes and methods

---

Reusability & Extensibility

- New borrow rules (e.g., faculty, event-specific loans) can be added simply by implementing a new BorrowStrategy subclass.
- The borrow logic is reusable across both the student and admin panels, decoupled from UI concerns.

GitHub Traceability

- Commits include [Pattern: Strategy] tags in messages.
- Strategy pattern applied in commits by **Mehmet Karatekin**.
- Documentation commits handled by **Aylin Barutçu**.

User Interface Design

This section describes the primary UI components supporting the **“Borrow Equipment”** use case. The goal is to ensure a smooth and intuitive borrowing experience with clear access to equipment listings, borrow forms, and confirmation feedback.

Key Screens

Equipment Catalog Page

- **Function:** Displays available equipment with filters (category, availability).
- **Features:**
  - Category dropdown

- Search bar

- Status badge (Available / Borrowed)

Equipment Detail Page

- **Function:** Shows individual equipment details, including description and condition.
- **Features:**
  - “Borrow Now” button (visible if item is available)
  - Estimated return date info

Borrow Request Form (Modal or Page)

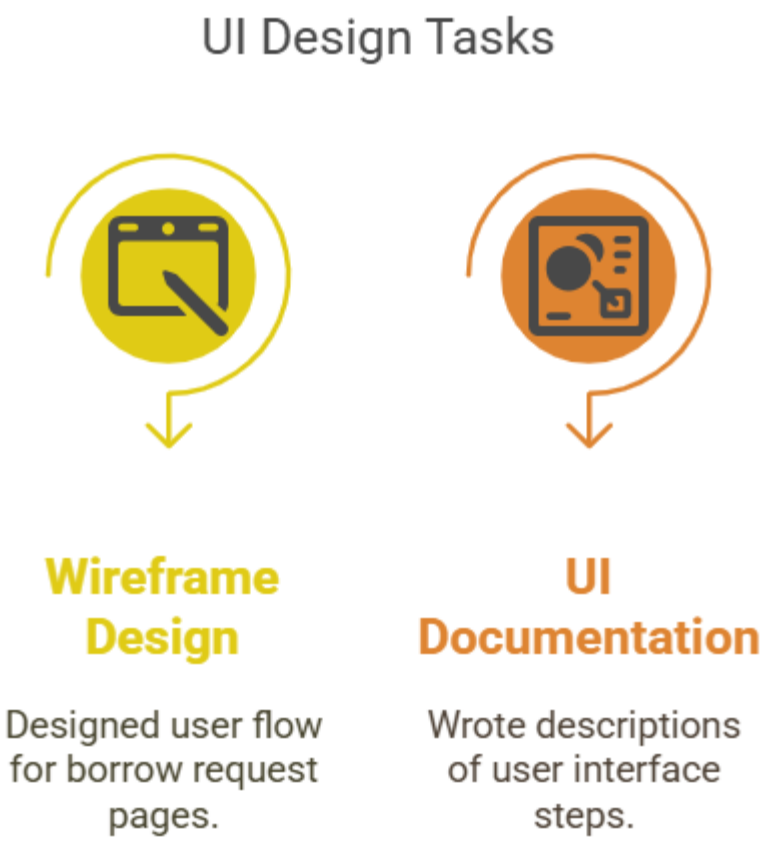
- **Function:** Collects information from the user for a borrow request.
- **Fields:**
  - Start Date
  - Return Date
  - Notes (optional)

Borrow Confirmation Page or Popup

- **Function:** Confirms the request has been submitted and provides status tracking link.

UI Mockup

A wireframe showing the interaction from catalog → detail → borrow form → confirmation has been generated.



Design Principles Followed

- **Minimal click depth:** All borrow actions reachable within 2–3 clicks.
- **Clean layout:** Focus on usability for non-technical students.
- **Accessibility:** High contrast text and large buttons for usability on mobile and desktop.
- **Responsive Design:** Mobile, tablet, and desktop compatibility.

External Interfaces

In the current MVP implementation of **KitHub**, the “Borrow Equipment” use case does not depend on real-time interaction with third-party systems. However, the architecture is designed to support external integration in future versions.

Current Status

- No active external interfaces in MVP.

- All logic and data processing are handled internally through the backend service and MySQL database.

Future Integration Possibilities

Interface Type	Description
University Login API	OAuth2-based SSO integration for student identity verification.
RFID/QR Tracking System	External inventory tools to scan and validate equipment movements.
Email Gateway (SMTP API)	External service for sending borrow confirmation and reminders.

These integrations are not implemented yet but were considered in the architectural decisions (such as modular REST endpoints and NotificationService separation).

Design Implication

Thanks to the layered architecture, any of the above systems can be connected in the **Application Layer** or via **service injection**, without modifying existing controller logic or database schema.

Performance Considerations

The performance expectations for the “Borrow Equipment” use case in **KitHub** are as follows:

Performance Metrics

Metric	Target Value
Page Load Time	Under 2 seconds on a standard broadband connection
API Response Time	Under 1 second for borrow request submission
Concurrent Requests	Up to 50 simultaneous users without degradation
Borrow Logic Execution	Strategy selection and processing < 200ms

These values were selected to support a mid-sized university user base and provide a smooth, responsive experience.

Scalability Strategy

- **Stateless Backend**:
  Designed with RESTful APIs, enabling horizontal scaling using container orchestration tools (e.g., Docker + Kubernetes).
- **Database Optimization**:
  - Use of indexes on EquipmentID and Status
  - Optimized SELECT queries with proper joins
- **Asynchronous Processing (Future Scope)**:
  - Notification emails or reminders to be processed in background queues (e.g., Celery or RabbitMQ)

Performance Testing Strategy

- Manual testing conducted with simulated concurrent user sessions
- Use of tools such as **Postman** for endpoint latency tracking
- **Chrome Lighthouse** audits for frontend speed feedback

Error Handling and Logging

Exception Management

To ensure system stability and provide clear feedback to users, all critical operations—especially those involved in the borrowing process—are wrapped in structured error handling. Key areas addressed include:

- Invalid equipment ID or unavailable equipment
- Incomplete borrow request forms
- Database connection issues
- Unauthorized access attempts

Logging Mechanisms

Every borrow request is logged with the following details:

- Timestamp

- User ID
- Equipment ID
- Borrowing strategy used

Errors and warnings are recorded in a rotating log file to maintain traceability and enable efficient debugging.

Monitoring & Debugging

- **Frontend:** All validation errors are shown inline in the borrow form for immediate user feedback.
- **Backend:** Logged errors are stored in files and may later be integrated with monitoring tools like Loggly or the ELK Stack for advanced tracking and analysis.

Test Cases

Test ID	Description	Acceptance Criteria	Definition of Done	Expected Result	Status
TC_AUTH_01	Register as non admin	A user can create a non admin account with username & password	Form submits via POST, UserFactory creates StudentUser, ClubUser redirected to /catalog	Non admin user is created and redirected	Passed
TC_AUTH_02	Register as admin	A user can create an admin account	UserFactory creates AdminUser, role and type are correctly assigned	Admin user is created and redirected	Passed
TC_AUTH_03	Login with valid credentials	User with correct credentials is authenticated	Password checked with check_password(), session established via Flask-Login	Redirect to /catalog	Passed
TC_AUTH_04	Login with invalid credentials	Login fails gracefully	Invalid credentials show error message in template	Login form reloads with error	Passed
TC_AUTH_05	Logout functionality	Clicking logout ends session and redirects to login	logout_user() clears session	Redirected to /login	Passed
TC_SEARCH_01	Search by name (e.g., "Cam")	Equipment with matching name should appear within 2 seconds	Name-based filtering is applied using decorator and tested successfully	Only equipment whose name includes "Cam" is shown	Passed

TC_SEARCH_02	Search by category (e.g., "audio")	Equipment matching selected category is returned correctly	CategoryFilterDecorator integrated, verified with various input values	Only equipment belonging to "audio" category is displayed	Passed
TC_SEARCH_03	Combine search with availability filter	Multiple filters should work together dynamically	Availability and search filters stack modularly using decorators; output matches filtered criteria	Only available equipment matching the search term is listed	Passed
TC_SEARCH_04	Leave search empty	All available equipment should be displayed by default	BaseEquipmentSearch returns available items without additional filters	All available equipment is shown	Passed
TC_ADM_01	Verify dashboard statistics	Dashboard cards must display real-time counts of users, equipment, and available items	Admin home route queries User and Equipment models, passes counts to template	Stats shown match database counts	Passed
TC_INV_01	Admin adds a new equipment item	New item must be created via form and appear in inventory	InventoryManager's add_equipment triggers Observer Pattern's notify method	A success flash message appears, the new item is visible in the table, and a 'EQUIPMENT_ADDED' log appears in the server console.	Passed

TC_INV_02	Admin changes item status to 'Maintenance'	Status must be updated in DB and UI must reflect change	The update_equipment_status route correctly handles form submission and calls the InventoryManager service.	The item's status in the table changes to 'Maintenance' and a 'STATUS_UPDATED' log appears in the console.	Passed
TC_OFR_01	Admin views pending borrow request	Pending requests must show 'Accept' and 'Reject' buttons	The inventory_list controller correctly queries for pending offers and passes the data to the template for conditional rendering.	Accept/Reject buttons are visible for the specific item with a pending offer, and 'No offers' is shown for all other items.	*Passed*
TC_OFR_02	Admin accepts borrow request	Accepting offer marks item as 'borrowed', request as 'approved'	The accept_offer route correctly updates both the BorrowRequest and Equipment tables in a single logical transaction.	Item status changes to 'borrowed', a success flash message appears, and the Accept/Reject buttons are removed for that item.	*Passed*
TC_OFR_03	Admin rejects borrow request	Rejecting offer marks request as 'rejected'	The reject_offer route correctly updates the BorrowRequest table without modifying the Equipment table.	Item status remains 'available', a rejection flash message appears, and the Accept/Reject buttons are removed.	*Passed*
TC_BOR_01	Student borrows available item	Student submits borrow request for available item	Strategy resolves to StandardBorrowStrategy, creates BorrowRequest with 7-day duration	BorrowRequest is saved with pending status, equipment remains available until approval	*Passed*

TC_BOR_02	Club member borrows available item	Club user submits borrow request for available item	Strategy resolves to ClubBorrowStrategy, creates BorrowRequest with 14-day duration	BorrowRequest is saved with pending status, duration reflects role-specific rule	*Passed*
TC_BOR_03	Borrowed item hidden from catalog	After a borrow request is approved, item should disappear from catalog	Catalog uses AvailabilityFilterDecorator to exclude non-available items	Borrowed equipment does not appear in /catalog listing	*Passed*
TC_BOR_04	Duration displayed on borrow form	Student or club visits borrow form	Strategy is resolved and max_duration is shown dynamically	Page shows correct duration (7 or 14) before submission	*Passed*
TC_BOR_05	Borrow strategy switch logic	Switching user roles should change borrow logic	BorrowController uses get_strategy_for_user() to resolve correct strategy	Borrowing durations and permissions adjust by role	*Passed*

Deployment and Installation Design

The system is deployed as a web-based application hosted on a local or cloud server.
The “Borrow Equipment” use case is integrated within the main backend service and available under the /borrow API route.

---

1. Environment Configuration

Component	Configuration
Backend	Python 3.11, Flask, MySQL
Frontend	HTML5, CSS3, JavaScript

Database	MySQL 8.x (initial local setup via SQLite)
OS Support	Cross-platform: Windows, macOS, Linux

---

2. Installation Instructions

1. **Clone the repository** ```bash git clone https://github.com/YourTeam/KitHub.git cd KitHub
2. **Create virtual environment** ```bash python3 -m venv env source env/bin/activate # On Windows: .
3. **Install dependencies** ```bash pip install -r requirements.txt
4. **Run database migration (for SQLite testing)** ```bash python scripts/init_db.py
5. **Run the application** ```bash flask run

---

3. Packaging & Structure

Folder	Purpose
/src/borrow/	Borrow logic using Strategy pattern
/docs/	Diagrams, UI mockups, and documentation

---

4. Future Hosting Options

- **GitHub Pages** (Frontend only)
- **Render / Heroku** (Full stack deployment)
- **Docker-based deployment** with Dockerfile (planned)

Change Log

Version	Date	Author	Change Description
1.0	28.06.2025	All team members	Initial draft of SDD created and reviewed
1.1	29.06.2025	İlbey Efe Taşabatlı	Updated strategy pattern section and DFD diagram
1.2	01.07.2025	Aylin Barutçu	Completed deployment instructions and setup notes
1.3	01.07.2025	Mehmet Karatekin	Added Key Features and Functionality section
1.4	01.07.2025	Mehmet Karatekin	Added Component Design section
1.5	02.07.2025	Mehmet Karatekin	Added design patterns and testability parts

---

Future Work / Open Issues

Known Open Issues

- No real-time inventory locking mechanism; may cause race conditions in concurrent borrow attempts.
- Lack of automated notifications (email/SMS); only internal alerts exist in current build.
- Minimal error feedback in UI (e.g., no toast messages, no modal confirmations).

Potential Enhancements

- Integration with university SSO systems (e.g., OAuth2)
- RFID/QR inventory scanning
- Return item damage reports
- Mobile-responsive UI improvements
- Admin analytics panel for borrow trends and stock usage

Task Matrix

Task	Description	Responsible Member
System Overview	Project introduction and use case context	Aylin Barutçu
System Context + UML Diagram	Use case boundary and actor diagram	İlbey Efe Taşabatlı
Key Features & Functionality	Explanation of how system meets use case goals	Mehmet Karatekin
Component Design + Diagram	Subsystems, responsibilities and visuals	Mehmet Karatekin

Data Design + ER & DFD	Entity modeling, DFD creation	İlbey Efe Taşabatlı
Design Pattern Integration	Strategy pattern logic, context, and usage	Aylin Barutçu
UI Design & Mockups	Interface logic and sketches	Betül Biçer
Performance & Error Handling	Performance goals, exceptions, and logging	Mehmet Karatekin
Deployment	How the system will configured and deployed	İlbey Efe Taşabatlı
Change Log	Versioning of design process	İlbey Efe Taşabatlı
Future Work & Open Issues	What's left, known bugs, and future plans	Betül Biçer

Measurable Acceptance Criterion

Test ID	Acceptance Criterion	Metric Formula	Target Value

MAC_SEARCH_001	Search results should match name or category with at least 90% accuracy.	(Correct results shown / Expected correct results) × 100	≥ 90%
MAC_PERF_001	Search and filter operations must return results within 2 seconds.	Average response time in seconds	≤ 2 seconds
MAC_AUTH_001	Newly registered users must be able to log in on first try.	Newly registered users must be able to log in on first try.	≥ 95%
MAC_AUTH_002	Borrow must be restricted to users with student and club roles only.	(Blocked borrow attempts by non-students / total such attempts) × 100	100%
MAC_AVAILABILITY_001	Unavailable items must not allow borrow action.	(# of unavailable items without borrow action / total unavailable items) × 100	100%
MAC_FILTER_001	Selected category filter must return only items matching that category.	(Items correctly matching selected category / Total items returned) × 100	≥ 95%
MAC_SUCCESS_RATE_001	Borrow requests for available items should succeed without error.	(Successful borrow requests / Total borrow attempts for available items) × 100	≥ 98%

MAC_ADM_KPI_001	The system must enable admins to respond to user borrow requests within the established service level agreement.	Average Time = Sum of (Resolution Timestamp - Creation Timestamp) / Total Completed Requests	≤ 4 business hours
MAC_ADM_REL_001	The system must enable admins to respond to user borrow requests within the established service level agreement.	Prevention Rate = (Successfully Blocked Invalid Actions / Total Attempted Invalid Actions) × 100	%100
MAC_ADM_UX_001	Core administrative actions (add, update, accept, reject) must complete within a timeframe that ensures a fluid user experience.	Average Transaction Time = Average of (Visual Page Update Timestamp - Button Click Timestamp)	< 1500 ms
MAC_BOR_001	Strategy selection (get_strategy_for_user) must return correct strategy based on role.	(# of correct strategy resolutions / total strategy resolutions) × 100	100%
MAC_BOR_002	Admin-initiated approvals must only apply to valid, strategy-compliant requests.	(# of approved requests that match original strategy constraints / total approved requests) × 100	≥ 98%
MAC_BOR_003	BorrowStrategy must execute eligibility check before saving to DB.	(# of borrow attempts checked via is_eligible() / total borrow attempts) × 100	100%