

CS 3354.502 Software Engineering

Final Project Deliverable 2

Textbook.io

Group Members

1. Nicholas Ackley
2. Zach Leach
3. Emily Allen
4. Kyle Haddad
5. Omar Hilweh
6. Mawi Mekonen
7. Jerin Vandannoor
8. Arjun Vishal

1. Final delegation of tasks

Nicholas Ackley

Deliverable 1: Report content formatting

Deliverable 2: Project scheduling, cost, effort, etc., Slides

Zach Leach

Deliverable 1: Tasking, Coordination, GitHub, Report content formatting

Deliverable 2: Tasking, Coordination, GitHub, Report content formatting, Conclusion, IEEE citations

Emily Allen

Deliverable 1: Response to feedback

Deliverable 2: Presentation prototype, Comparison to similar designs

Kyle Haddad

Deliverable 1: Software model chosen, Architectural design

Deliverable 2: Slides

Omar Hilweh

Deliverable 1: Class diagram

Deliverable 2: Slides

Mawi Mekonen

Deliverable 1: Non-functional requirements

Deliverable 2: Slides

Jerin Vandannoor

Deliverable 1: Functional requirements

Deliverable 2: Unit testing, Slides

Arjun Vishal

Deliverable 1: Use cases, Sequence diagrams

Deliverable 2: Conclusion, Slides

2. Deliverable 1 Content

CS 3354.502 Software Engineering

Final Project Deliverable 1

Textbook.io

Group Members


9. Nicholas Ackley
10. Zach Leach
11. Emily Allen
12. Kyle Haddad
13. Omar Hilweh
14. Mawi Mekonen
15. Jerin Vandannoor
16. Arjun Vishal

1. Project Goals, Motivation, and Task Delegation

Project Goal: Create a textbook application that targets students in college. Students who have completed a course can buy, sell, or trade books to underclassmen in need of this textbook for a used condition, and at a cheaper price.

Motivation: We chose to create an app for selling, renting, and buying books because it's easy to implement, and there is a demand for a service like this. This application is made to help students and other users find the books they are looking for at a better price and sell or rent their old books.

Proposal Feedback: N/A.

A graphic showing a grey bar with the text "Project Proposal" and a white box below it containing the text "Well done."

Project Proposal

Well done.

References: Sommerville, Ian. *Software Engineering*. Ed. 10th Boston: Pearson, 2016.

2. GitHub

GitHub repository URL: <https://github.com/zcl190002/3354-Group1>.

3. Delegation of Tasks for Deliverable #1

- Member Responsibilities:
 - Nicholas Ackley: Tasked with compiling individual contributions into the final report.
 - Zach Leach: Tasked with github setup and member coordination.
 - Emily Allen: Tasked with responding to feedback from Project Proposal.
 - Kyle Haddad: Tasked with writing justification for software process model chosen and assumptions.
 - Omar Hilweh: Tasked with creating class diagrams.
 - Mawi Mekonen: Tasked with writing non-functional requirements.
 - Jerin Vandannoor: Tasked with writing functional requirements.
 - Arjun Vishal: Tasked with writing use cases and creating sequence diagrams.

4. Software Process Model

The software process model employed for this project is Waterfall. The initial phases of the Waterfall Model, namely Requirement Analysis, Design, and Implementation, closely align with the requirements for the project deliverables. Additionally, the Waterfall Model prioritizes documentation, which allows group members to track tasks and communicate with each other.

5. Software Requirements

User Registration and Authentication: The system should allow new users (students) to register for an account, including email verification and secure password setup. Registered users should be able to log in and access their profiles.

Textbook Listing Management: Users (sellers) should be able to list textbooks they want to sell, including details like title, author, edition, price, and condition. They should also be able to manage (edit, update, or delete) their listings.

Textbook Search and Filter: The application should provide a search feature where users can find textbooks based on keywords like title, author, edition, or course code. Filters (e.g., price range, condition, or availability) should be available to narrow down results.

Transaction and Payment Processing: The system should enable secure transactions between buyers and sellers. This includes adding items to a cart, providing payment methods, processing payments, and confirming purchases.

Communication and Reviews: The system should allow users to communicate, such as messaging between buyer and seller for queries about a listing. Additionally, buyers should be able to leave ratings and reviews for sellers after transactions are completed.

Book Return and Dispute Handling: The platform should support functionality for initiating returns and resolving disputes between buyers and sellers when needed.

6. Non-Functional Requirements

Dependability

- The system shall ensure high availability and reliability, with minimal downtime for maintenance.
- The application shall have backup mechanisms to recover data in case of system failure.

Security

- Product- The system shall implement secure authentication methods, including password encryption. The application shall use secure payment gateways to protect users' financial

information during transactions. The system shall provide measures to prevent unauthorized access and protect user accounts.

- Legislative - The application shall comply with data protection regulations to safeguard user information.

Regulatory

- The system shall adhere to relevant laws and regulations concerning e-commerce and data privacy.
- The application shall provide users with transparent policies regarding data usage and user rights.

Ethical

- The system must allow users to report unethical behavior, such as fraudulent listings or scams.
- The system shall consider the environmental impact of hosting and data storage solutions and aim for sustainable practices.

Usability

- The application shall provide an intuitive user interface that allows easy navigation for buyers, sellers, and renters, ensuring users can quickly find the features they need.

User Account Management

- The system shall allow new users to sign up, creating an account and sending a confirmation.
- Registered users shall log in to view their profiles, history, and wishlist.
- The system shall enable sellers/renters to log in, upload book details, confirm listings, and have the system validate and display these listings.
- The system shall allow buyers/renters to initiate returns or disputes. It shall track returns, enabling sellers to resolve these issues efficiently.

Books

- The system shall allow users to search for books by title, author, genre, ISBN, and keywords and filter the results by price, publication, and customer rating.
- The system shall allow users to rent books by selecting a rental option on the product page, specifying the rental duration, and providing payment details. The system will

automatically calculate and display the due date and rental fees based on the selected duration.

- The system shall allow registered users to leave comments and reviews on individual book product pages, with the ability to rate the book on a scale of 1 to 5 stars and view other users' comments sorted by most helpful or most recent.

Wishlist

- The system shall allow users to add books to a personal wishlist, view and manage their wishlist items, and move books from the wishlist to the shopping cart for future purchase. The system shall also notify users if a book in their wishlist goes on sale or becomes available if previously out of stock.
- The system shall facilitate buyers/renters in adding books to their shopping carts, modifying the quantity, proceeding to checkout where they can enter the payment details and shipping information to complete the purchase, and updating both parties involved in transactions.
- The application shall be optimized for both desktop and mobile devices, ensuring a seamless experience across different platforms.

Environmental

- The system shall promote sustainable practices, such as digital documentation, to reduce paper usage.
- The application shall consider energy efficiency in its operations and infrastructure.

Operational

- The application shall maintain performance metrics to assess user satisfaction and system efficiency.
- The system shall be capable of handling simultaneous user interactions without degradation of performance, especially during peak usage.
- User actions, such as book listing uploads and payments, must be processed promptly to enhance user satisfaction.
- The system shall be able to operate on various platforms, including web and mobile applications, providing a seamless experience across devices.

Development

- The application shall follow best practices in software development, including code reviews and testing protocols.
- The system shall be designed using the Model-View-Controller (MVC) architecture to facilitate modular development and ease of updates.
- The application shall allow for easy integration with third-party services, such as payment gateways and eBook providers.
- The system shall allow for future scalability and the addition of new features based on user feedback.

Performance

- The system shall ensure quick response times for user actions, aiming for less than 3 seconds for loading pages and processing requests.
- The application shall support simultaneous access by multiple users without degradation in performance.
- The system shall maintain optimal loading times for pages and search results, aiming for less than 3 seconds for most interactions.
- The application shall effectively manage database queries to minimize latency and ensure efficient data retrieval.

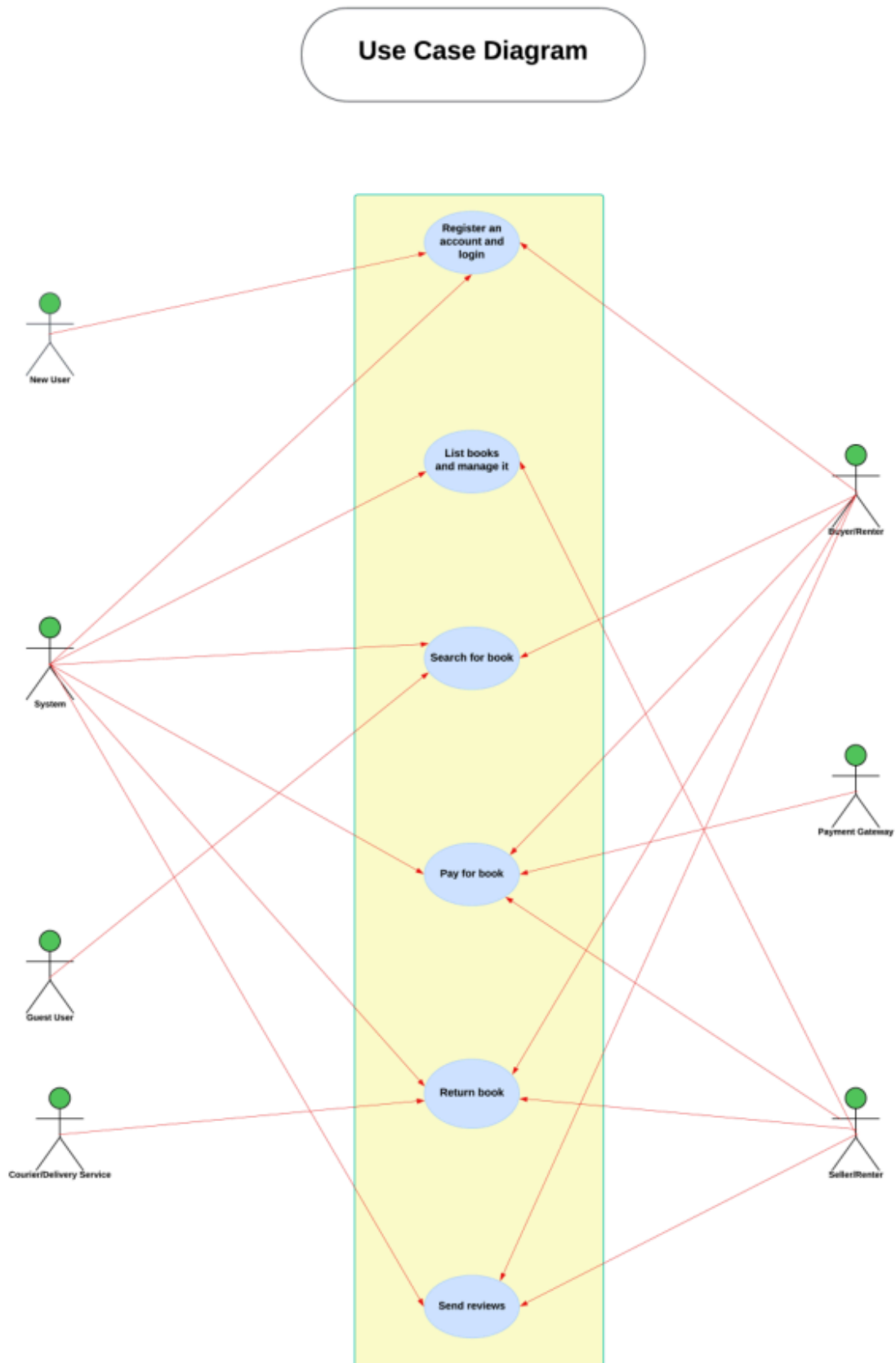
Space

- The application shall efficiently utilize server space, optimizing database storage to handle user data and transactions.
- The system shall implement regular database maintenance procedures to ensure optimal performance.

Accounting

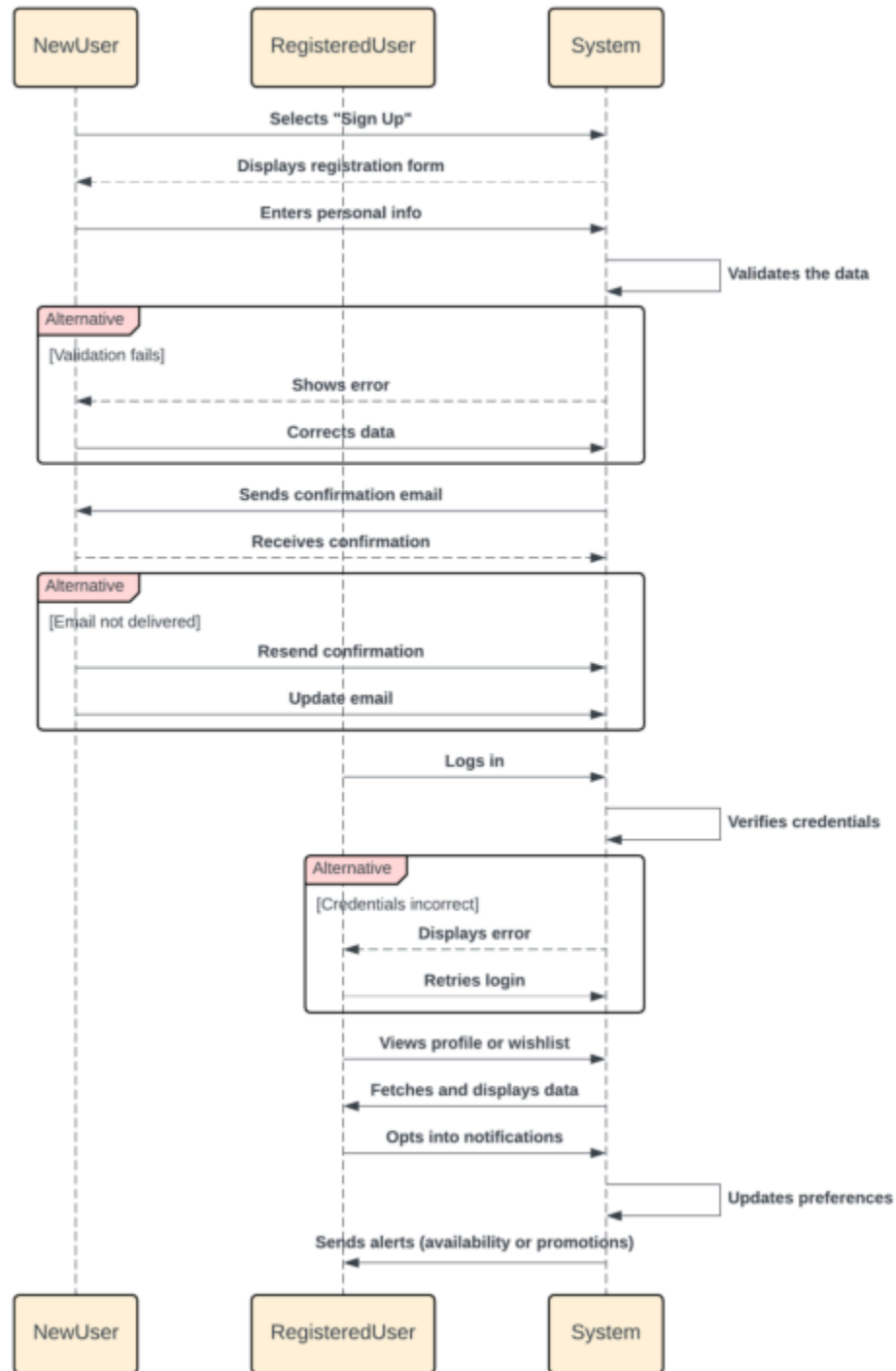
- The system shall provide detailed transaction histories for users, allowing them to track purchases and rentals.
- The application shall ensure accurate calculations of fees, taxes, and discounts during transactions.
- Users must receive clear, itemized receipts for purchases and rentals.

7. Use Case Diagram

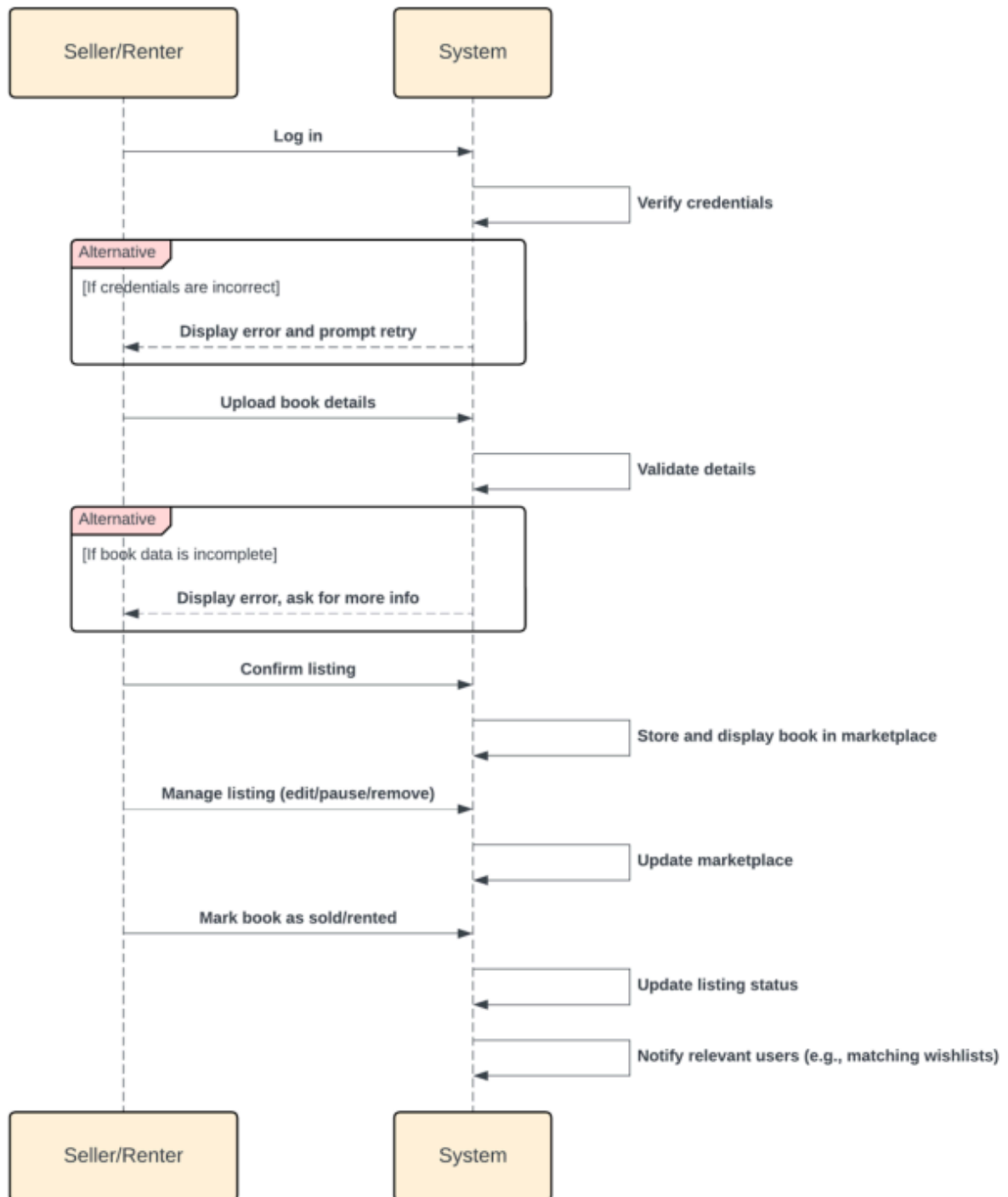


8. Sequence Diagrams

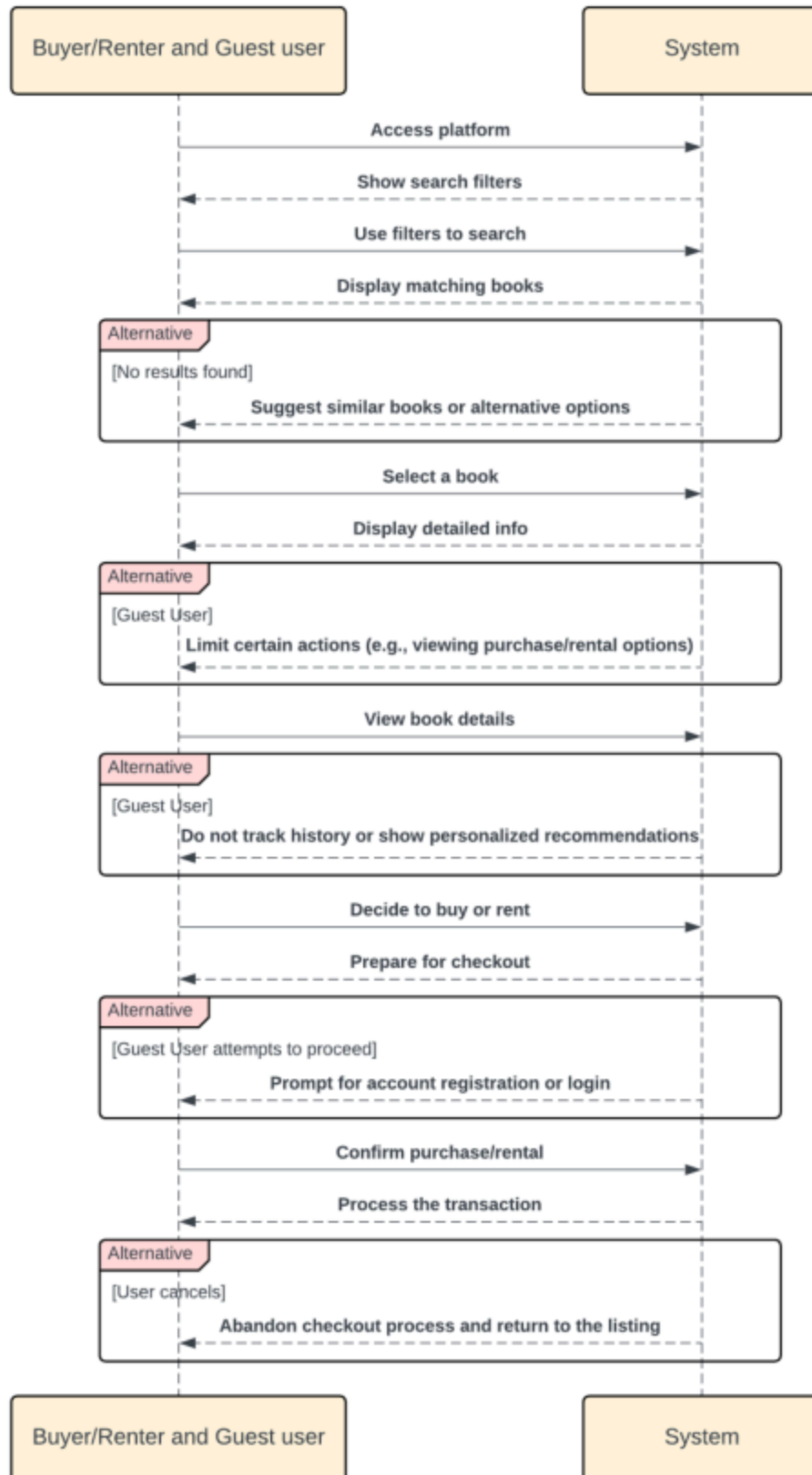
1) Register an account and login



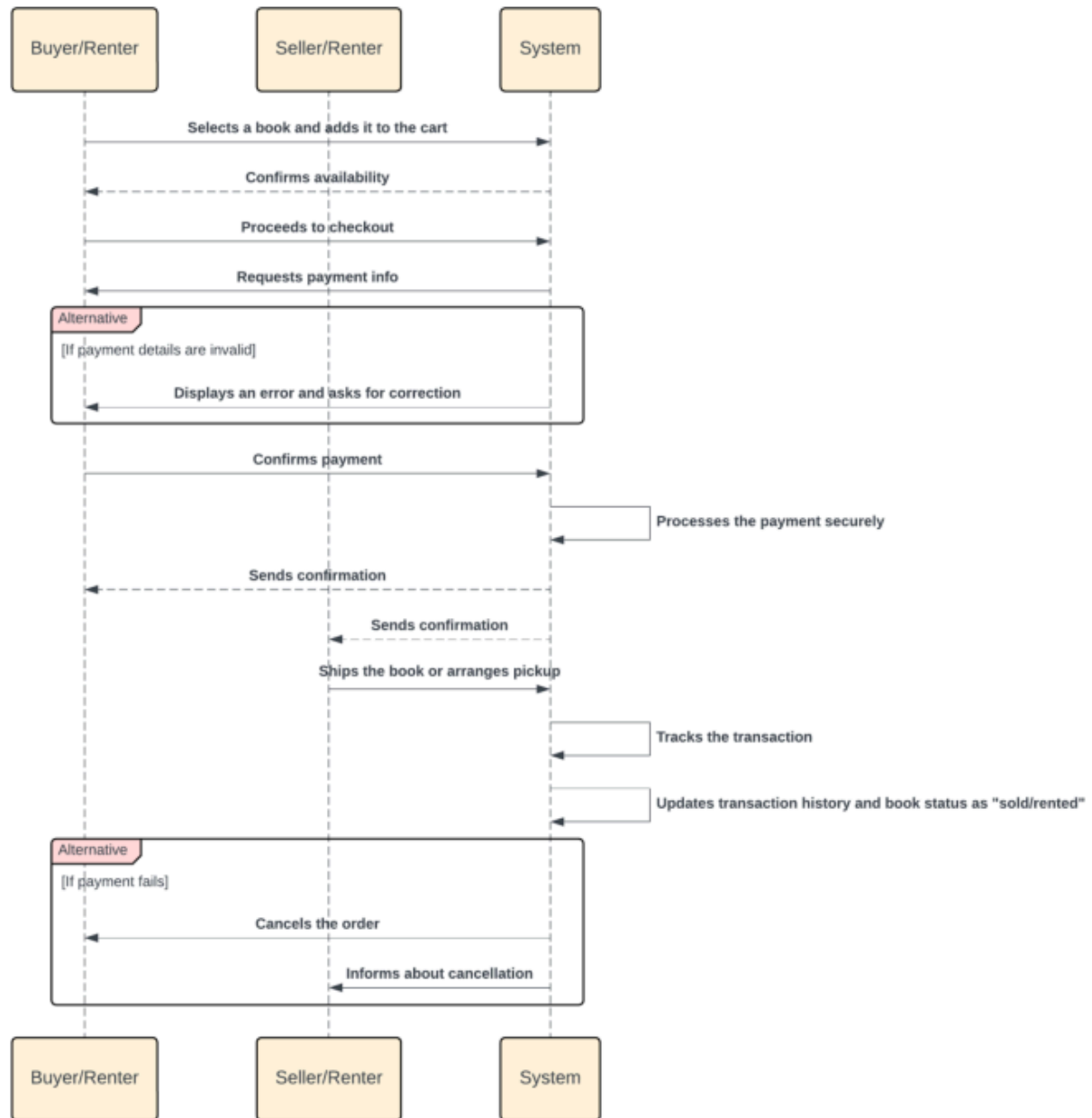
2) List books and manage it



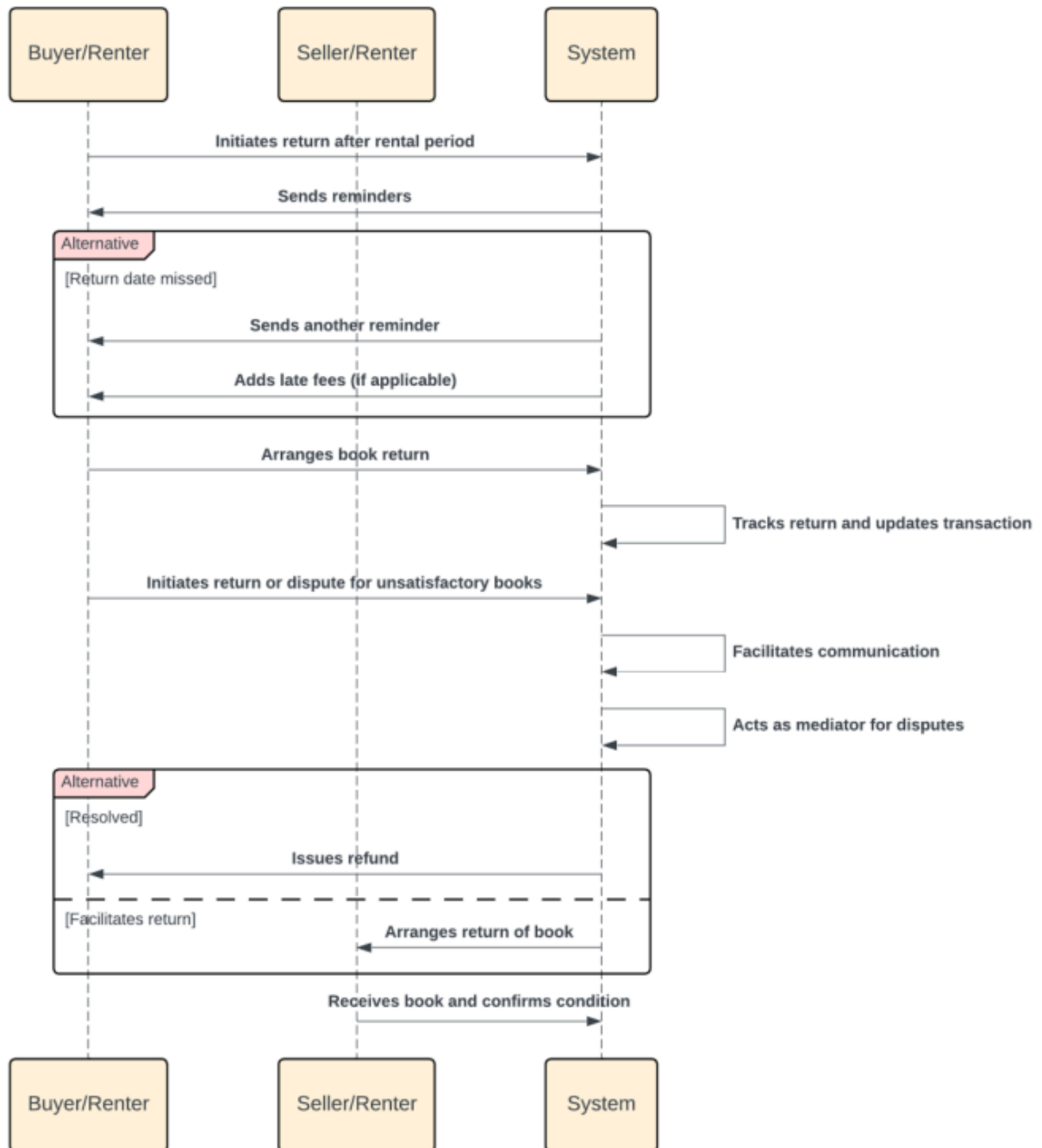
3) Search for book



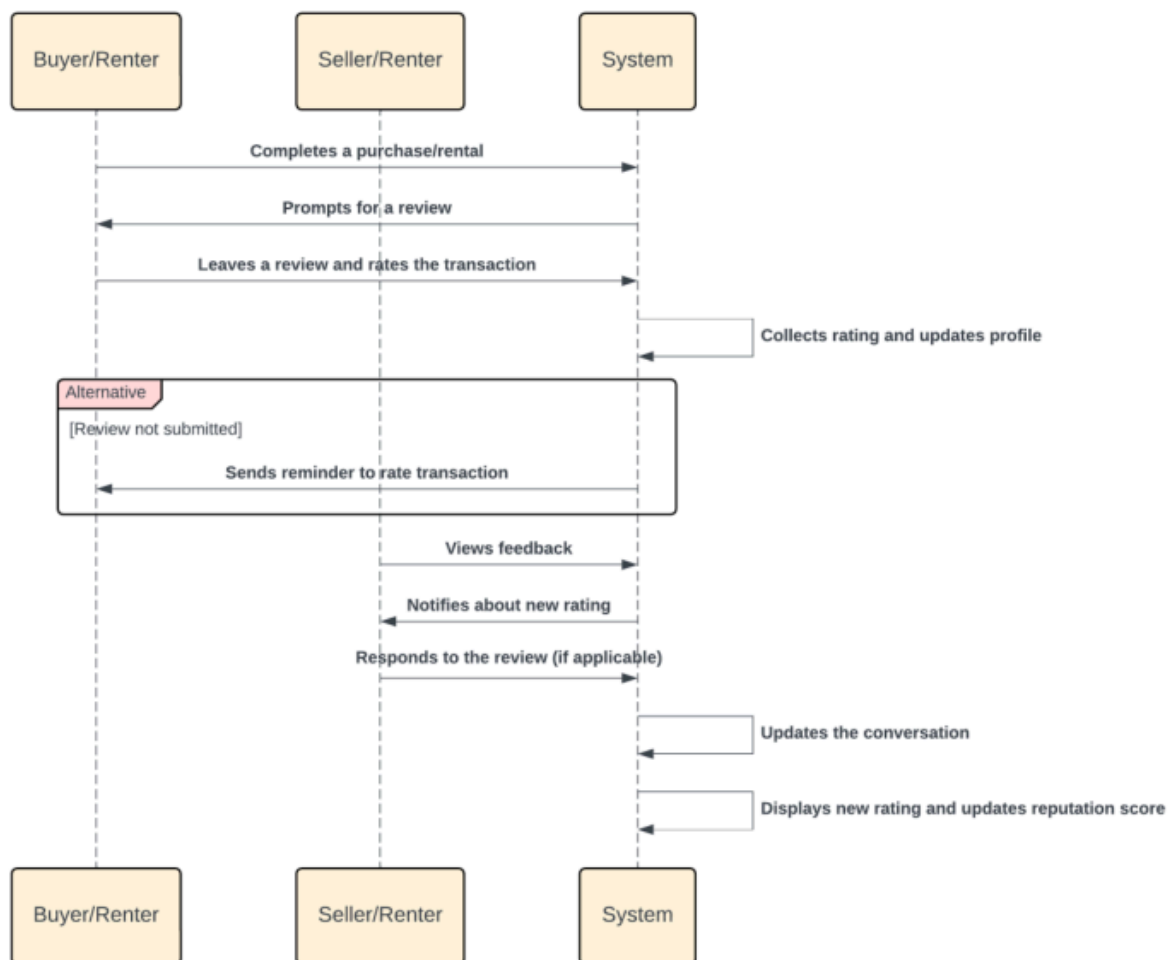
4) Pay for book



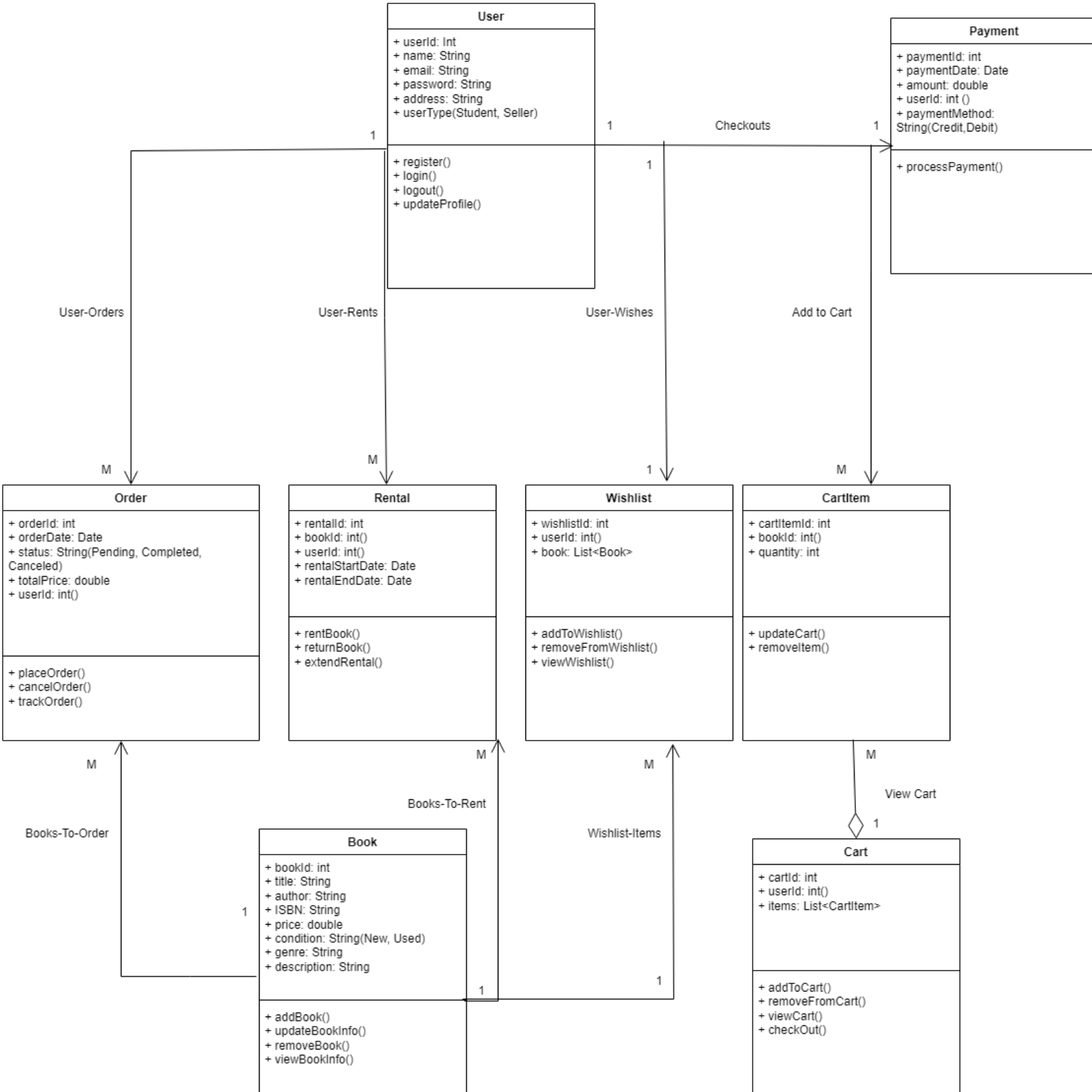
5) Return book



6) Send reviews



9. Class Diagram



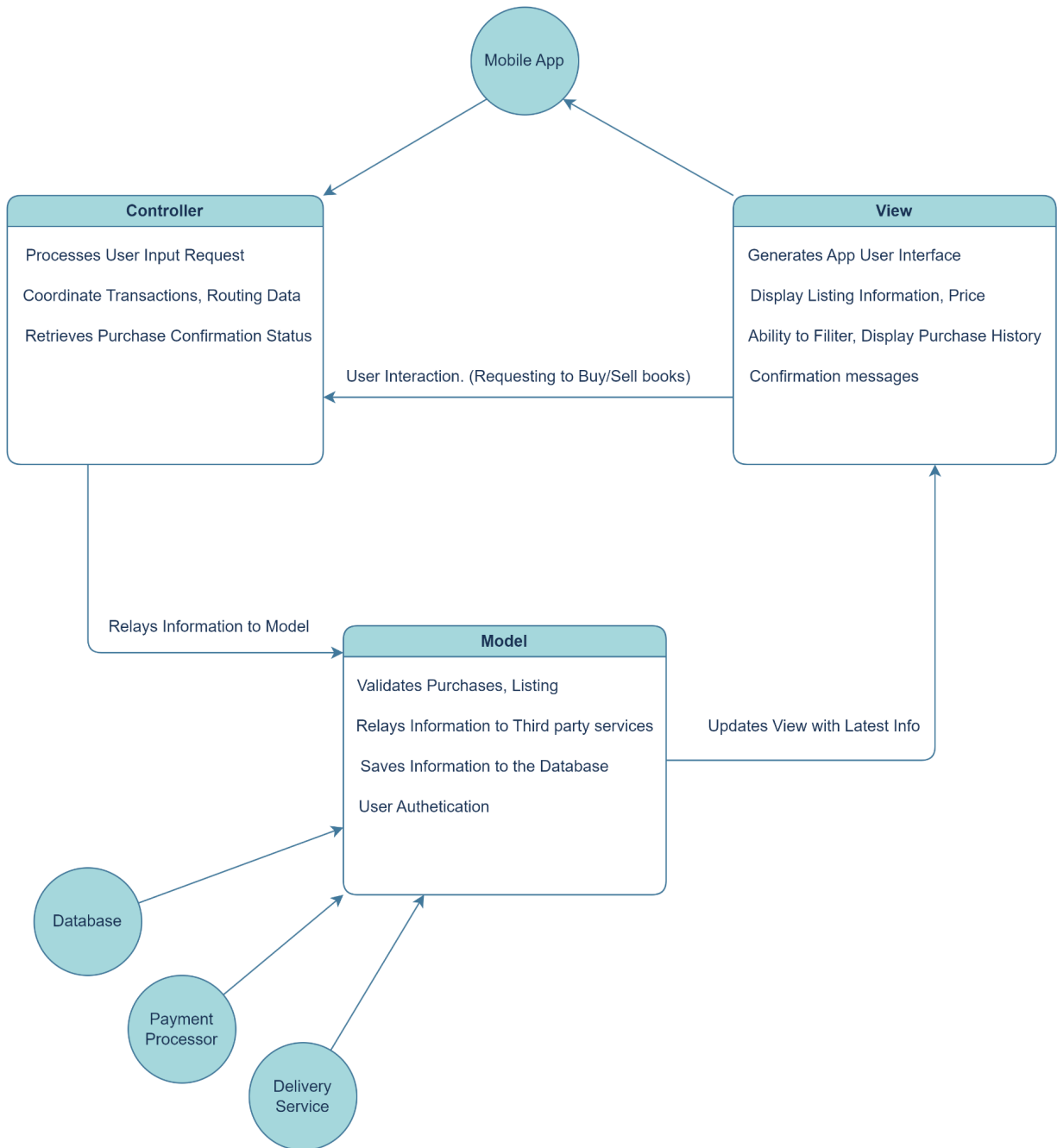
10. Architectural Design

We chose the Model-View-Controller (MVC) architecture pattern for our textbook selling application, because of its flexibility, ease of scalability, and efficient handling of server communication.

The platform will need to support a wide variety of user roles, such as sellers, renters, and partnered book suppliers. Each of these users have different needs and would require different interfaces. MVC allows us to easily accomplish this because the View Component is completely separated from any other logic, allowing us to create unique views as needed. The Controller component provides a shared set of modules for managing the user input, and processing requests. This streamlines development of views as we don't have to recreate the same modules and logic for each view, rather they can be shared as needed.

Since our platform is a mobile application that relies heavily on user interaction, frequent communication occurs between the user-facing application and the backend database as users create listings, purchase products, and leave reviews. The MVC Architecture supports these interactions by allowing efficient direct communication between the View (User Interface) and the Model (Database). Additionally the centralized model is important for keeping consistently being users especially when purchasing a single product.

Scalability is important for our platform, as typically following the end of a school year we see a spike in demand as students sell off their old books, similarly at the beginning of school year a spike in purchasing. Because each component in MVC is separated you have the ability to have multiple instances of a Model, we can easily scale up and down various components as each one is put under strain by seasonal events. This improves overall system performance and stability, while also saving us money by reducing server costs when not needed.



This page is intentionally left blank

3. Project scheduling

Cost, Effort, and Pricing (Composition Model)

The application composition model, introduced as part of COCOMO II, specifically targets prototyping projects and software developed using existing components. The model calculates effort based on four fundamental counts:

1. Number of screens/web pages displayed
2. Number of reports produced
3. Number of modules in imperative programming languages (e.g., Java)
4. Number of lines of scripting/database programming code

The effort calculation involves a specific formula:

$$PM = (NAP \times (1 - \%reuse/100)) / PROD$$

Where:

- PM is the effort estimate in person-months
- NAP represents total application points in the system
- %reuse is the percentage of reused code expected
- PROD is the productivity rate, which varies based on two key factors:
 - Developer's experience and capability (ranging from very low to very high)
 - ICASE (Integrated Computer Aided Software Engineering) tool maturity

The productivity rate (PROD) ranges from 4 NAP/month for very low capability/maturity to 50 NAP/month for very high capability/maturity, with nominal projects achieving around 13 NAP/month.

The application composition model is the better choice for our textbook trading platform because it's specifically designed for projects that heavily utilize existing components and prototyping - which aligns perfectly with our modern web application needs. The model directly accounts for screens (like our login, book listing, and search pages), reports (transaction histories), and reusable components (authentication systems, payment processing modules) that make up the core of our application.

Furthermore, the model's explicit consideration of developer experience and ICASE tool maturity through the PROD rate makes it more relevant for our project's context, where we'll likely be using modern development frameworks and libraries. The formula $PM = (NAP \times (1 - \%reuse/100)) / PROD$ gives us a more realistic estimate by factoring in code reuse, which is significant in our case since we'll be using existing components for features like user authentication and payment processing. Function points, in contrast, would require more complex counting of abstract elements and wouldn't directly account for the component-based nature of our development approach.

Screens (Web Pages):

- Registration/Login pages (2)
- User Profile/Dashboard (1)
- Textbook Listing Creation/Edit (1)
- Search/Browse Page with Filters (1)
- Book Details Page (1)
- Shopping Cart/Checkout (2)
- Messaging Interface (1)
- Reviews/Ratings Page (1)
- Dispute/Return Management (1)

Total Screens: 10 screens

Reports:

- Transaction History
- Sales Reports
- User Activity Reports
- Dispute Resolution Reports

Total Reports: 4 reports

Modules (Java/Backend):

- User Authentication
- Textbook Management
- Search/Filter System
- Payment Processing
- Messaging System
- Review System
- Dispute Management

Total Modules: 7 modules

Database/Scripting Components:

- User Database Operations
- Textbook Listings Database
- Transaction Processing
- Message Storage
- Review/Rating Storage

Total Script Components: 5 components

Calculation:

1. Assuming moderate complexity for most components
2. Developer experience: Nominal (student project)
3. ICASE maturity: Low (typical for academic environment)
4. PROD rate (from table): 7 NAP/month
5. Expected code reuse: 30% (using existing libraries for auth, payments)

Total NAP (New Application Points) = 26 (sum of all components)

Using the formula:

- $PM = (NAP \times (1 - \%reuse/100)) / PROD$
- $PM = (26 \times (1 - 30/100)) / 7$
- $PM = (26 \times 0.7) / 7$
- $PM = 18.2 / 7$
- $PM \approx 2.6$ person-months

Assuming an average software developer cost of \$6000 per month:

Estimated Cost = 2.6 person-months \times \$6000

Estimated Cost \approx \$15,600

Schedule Planning

Start Date: January 15, 2025

End Date: March 31, 2025 (approximately 2.5 months)

Working Parameters:

- Working days: Monday through Friday (weekends excluded)
- Working hours: 8 hours per day (standard business hours 9 AM - 5 PM)
- Total working days per month: ~22 days
- Total working hours per month: ~176 hours

Justification:

The 2.6 person-months calculation translates to approximately 11 weeks of development time. We've also added a small buffer for:

- Initial project setup and team onboarding (3 days)
- Integration testing periods (5 days)
- Deployment preparation (2 days)

The January 15th start date allows for proper project initialization after the holiday season, and the March 31st end date provides a realistic timeline considering:

1. The complexity of payment integration systems (typically takes 2-3 weeks)
2. User authentication and security features (1-2 weeks)
3. Database design and implementation (1-2 weeks)
4. UI/UX development for 10 screens (2-3 weeks)
5. Testing and bug fixing (2 weeks)

Weekends are excluded because:

1. This appears to be a standard business project
2. Weekend exclusion allows for team rest and reduces burnout
3. Provides buffer time for unexpected delays
4. Aligns with typical software development practices

The 8-hour workday is chosen because:

1. It's the standard industry practice
2. Maintains team productivity and focus
3. Allows time for daily meetings, code reviews, and documentation
4. Provides sustainable work-life balance for the development team

Hardware Costs:

- Production Server: \$1,200/year (AWS t3.large instance)
- Development/Testing Server: \$600/year (AWS t3.small instance)
- Database Server: \$1,000/year (AWS RDS instance)
- Load Balancer: \$300/year
- Storage (for book images/data): \$500/year (S3 storage)

Total Hardware: ~\$3,600/year

Software Products:

- SSL Certificate: \$200/year
- Payment Gateway License: \$600/year (Stripe standard plan)
- Development Tools/IDE Licenses: \$400/year
- Database Management System: \$500/year
- Monitoring/Analytics Tools: \$300/year

Total Software: ~\$2,000/year

Development Team (Personnel Costs):

- 1 Senior Developer: \$8,000/month
- 1 Junior Developer: \$5,000/month
- 1 Part-time UI/UX Designer: \$3,000/month
- Duration: 2.6 months (from our application composition calculation)

Development Cost: $(8,000 + 5,000 + 3,000) \times 2.6 = \$41,600$

Training and Support (Personnel Costs):

- Initial System Admin Training: \$2,000
- User Documentation: \$1,000
- Post-deployment Support (3 months): \$3,000

Total Training/Support: \$6,000

Total Project Cost Estimate:

Hardware: \$3,600

Software: \$2,000

Personnel: \$47,600

Total: \$53,200

4. Testing

We tested the user registration email and password input validation functionality of our application using the following 6 unit tests.

1. Valid Registration
 - a. input: valid email and strong password
 - b. expected output: true
2. Invalid Email
 - a. input: invalid email and strong password
 - b. expected output: false
3. Weak Password
 - a. input: valid email and weak password
 - b. expected output: false
4. Duplicate Email
 - a. input: valid email that already exists and strong password
 - b. expected output: false
5. Missing Email
 - a. input: empty string email and strong password
 - b. expected output: false
6. Missing Password
 - a. input: valid email and empty string password
 - b. expected output: false

app > src > main > java > org > example > UserRegistration.java

```
1  package org.example;
2
3  import java.util.HashSet;
4  import java.util.regex.Pattern;
5
6  public class UserRegistration {
7
8      private static HashSet<String> registeredEmails = new HashSet<>();
9
10     public static boolean registerUser(String email, String password) {
11         if (!isValidEmail(email)) {
12             System.out.println("Invalid email format.");
13             return false;
14         }
15
16         if (!isPasswordStrong(password)) {
17             System.out.println("Password does not meet strength requirements.");
18             return false;
19         }
20
21         if (isEmailInUse(email)) {
22             System.out.println("Email is already registered.");
23             return false;
24         }
25
26         registeredEmails.add(email);
27         System.out.println("Registration successful.");
28         return true;
29     }
30
31     private static boolean isValidEmail(String email) {
32         String emailRegex = "^[A-Za-z0-9+_.-]+@(.+)$";
33         Pattern pattern = Pattern.compile(emailRegex);
34         return pattern.matcher(email).matches();
35     }
36
37     private static boolean isPasswordStrong(String password) {
38         return password.length() >= 8 &&
39             password.matches(".*[A-Za-z].*") &&
40             password.matches(".*[0-9].*");
41     }
42
43     private static boolean isEmailInUse(String email) {
44         return registeredEmails.contains(email);
45     }
46
47     public static void clearRegisteredEmails() {
48         registeredEmails.clear();
49     }
50 }
```

app > src > test > java > org > example > UserRegistrationTest.java

```
1  package org.example;
2
3  import org.junit.jupiter.api.BeforeEach;
4  import org.junit.jupiter.api.Test;
5  import static org.junit.jupiter.api.Assertions.*;
6
7  public class UserRegistrationTest {
8
9      @BeforeEach
10     public void setUp() {
11         UserRegistration.clearRegisteredEmails();
12     }
13
14     @Test
15     public void testValidRegistration() {
16         assertTrue(UserRegistration.registerUser("valid@example.com", "StrongPass123"));
17     }
18
19     @Test
20     public void testInvalidEmail() {
21         assertFalse(UserRegistration.registerUser("invalid-email", "StrongPass123"));
22     }
23
24     @Test
25     public void testWeakPassword() {
26         assertFalse(UserRegistration.registerUser("user@example.com", "123"));
27     }
28
29     @Test
30     public void testDuplicateEmail() {
31         UserRegistration.registerUser("duplicate@example.com", "StrongPass123");
32         assertFalse(UserRegistration.registerUser("duplicate@example.com", "NewPass123"));
33     }
34
35     @Test
36     public void testMissingEmail() {
37         assertFalse(UserRegistration.registerUser("", "StrongPass123"));
38     }
39
40     @Test
41     public void testMissingPassword() {
42         assertFalse(UserRegistration.registerUser("user@example.com", ""));
43     }
44 }
```

5. Comparison of work with similar designs

There are two primary categories of applications that are similar to *textbook.io*: traditional marketplaces for the resale of books alongside general merchandise, and specialized platforms designed for students to buy, sell, and share textbooks with one another.

Examples of the former category include eBay and Facebook Marketplace. These platforms allow users to reach a broad audience, which can be advantageous in urban areas with multiple universities. However, the typical Facebook marketplace user may not be actively seeking textbooks, and eBay requires the seller to manage shipping logistics, which can be burdensome for busy college students. While Chegg facilitates the buying, selling, and rental of textbooks, it shares similar drawbacks to eBay or Amazon, where long-distance shipping incurs both financial costs and environmental impact.

Hence, there exists a need for a platform dedicated specifically to the local exchange of textbooks among college students. Several platforms have been developed for this purpose. Notably, an application named BonoBooks previously served students of the University of Washington, but this app is no longer available on the Apple or Google Play stores. During its operational period, BonoBooks incentivized students by offering \$7 for each book they listed, although the source of this funding remains unclear. It is plausible that revenue was generated through in-app advertisements, but users would likely become irritated if ads were too prevalent. In contrast, *textbook.io* proposes a model in which sellers receive compensation directly from the buyers, so the only cost to the developers would be hosting.

By confining textbook exchanges to specific university communities, students would be able to save money while also contributing positively to their peers and the environment. Textbooks utilized in one semester are likely to be used in subsequent semesters for the same courses. Should users opt to list their textbooks at no cost, they could alleviate the financial burden on students who may struggle to afford new textbooks. *Textbook.io* presents an opportunity to build campus communities, reduce the environmental impact of physical textbooks, and lessen the financial strain on college students.

6. Conclusion

We started by assigning tasks, but had to make changes close to the deadline when we discovered the grading rubric was posted in eLearning and was different from our tasking. This was unexpected and caused extra work. Additionally, giving everyone tasks to do over 4 weeks was a mistake; everyone waited until the last minute to finish. We should have split tasks into weekly steps instead. Had we adopted a more agile methodology and adopted weekly sprints into our team's workflow, we may have been able to subvert this.

Tasking is difficult; task descriptions need to be clear and easy for team members to accomplish. This means really understanding the tasks and explaining them well, which takes skill. One reliable person should handle task delegation (e.g., a Product Owner in agile methodologies). Having more than one person doing it makes things more complex because they have to talk to each other and wait for responses. We corrected this in the second deliverable by giving the team member responsible for tasking the job of compiling the final report, because they were more familiar with each task needed on the report.

Don't work on your task without confirming you understand what exactly is required (e.g., requirements verification and validation). You don't want to waste time working on the wrong thing. For instance, creating several diagrams only to be told you did all of them wrong and need to start over; luckily that didn't happen to us. We should've had team members confirm with the task coordinator that they correctly understood their task before getting started.

7. IEEE References

- [1] "Bonobooks," BonoBooks, <https://www.bonobooks.app/home> (accessed Nov. 3, 2024).
- [2] E. Kalliamvakou et al., "The Promises and Perils of Mining GitHub," in Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14), Hyderabad, India, 2014, pp. 92-101, doi: 10.1145/2597073.2597074.
- [3] B. Morgan, "How Airbnb's Focus on Usability Enhances Customer Experience," Forbes, Mar. 19, 2018. [Online]. Available: <https://www.forbes.com>. [Accessed: Nov. 10, 2024]
- [4] A. Cockcroft, "Performance and Reliability in Netflix Streaming: A Case Study," IEEE Internet Computing, vol. 20, no. 3, pp. 73-77, 2018.
- [5] J. Hamilton, "Internet Scale in the Cloud: Scaling to 10 Billion Requests Per Day and Beyond," Communications of the ACM, vol. 61, no. 8, pp. 92-101, Aug. 2018, doi: 10.1145/3232559.