Online Book Store Management System - Software Requirements & Implementation Specifications

User Requirements

The online bookstore system will streamline the process of browsing and ordering items for customers, while providing employees with tools to manage inventory and fulfill orders efficiently.

System should store information about its customers and employees. Every customer has name, surname, email, username, password, date of birth and address. For employees we need name, surname, email, username, password, salary, experience (it can be junior, middle or senior) and minimum salary possible which is 2000 (minimum salary cannot be decreased).

The store sells items: books, magazines, newspapers (there are also items without a type). Every item has name, description, image URL, publishing date, language, price and amount in stock. Also, they have minimum age. We want customers to be able to access only items that are appropriate to their age category. Each item has a publisher. We store name, address, email and phone number about them.

Items can be used, in that case we want to know is item has any annotations and its condition (mint, good, fair or poor). New items can be sealed or without any packaging.

If an item is a book. It can have genres and at least one author. Also, we want to know number of pages and cover type (hard, soft or spiral bound). For authors we need to store their name, surname, date of birth and, optionally, pseudonym. For magazines we distinguish special and non-special editions; newspaper has a headline and list of topics.

Customers will be able to browse the store’s catalog, view item details, and add specific number of items to a cart. If the cart contains at least one item, the customer can proceed to checkout or edit the cart. During the checkout process, they will be required to provide all necessary personal and shipping information. After completing checkout, the system will direct the customer to the payment stage to finalize the purchase. After that system will save the payment details (type of payment, time stamp, amount).

Customers will also have access to their order history. If customer paid for order but its preparation has not started yet they should be able to cancel it. Moreover, they should be able to view the details of any order, including the list of ordered items. From this list, they can select any item to view its details.

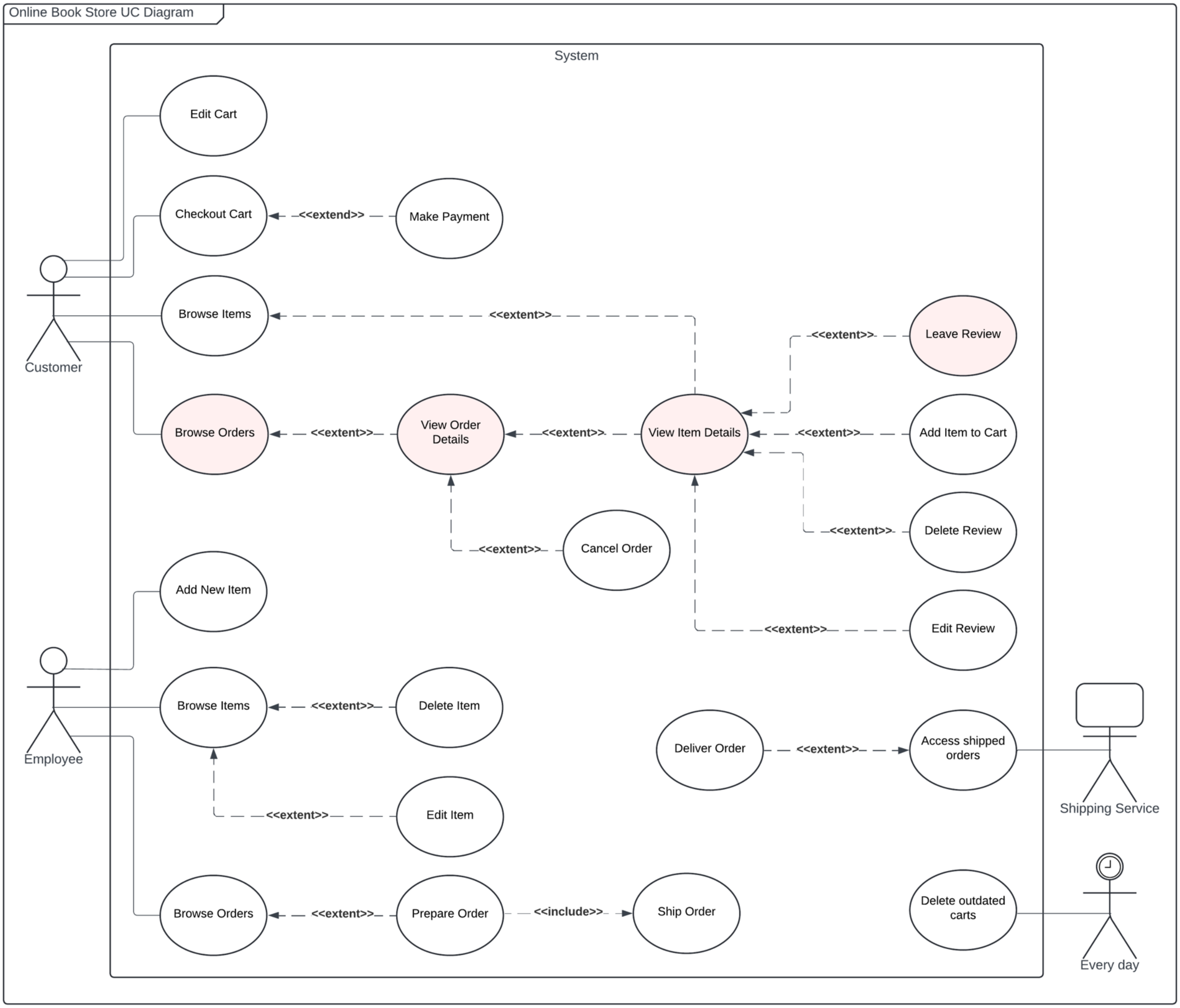
Additionally, customers should be able to leave reviews for items they have received. To do this, they must navigate to item details view through catalog or any of their orders. If the selected item has been delivered, a button to leave review will be visible on the item detail page. This option will not be available for items that have not yet been delivered. When the customer clicks the button, they will be taken to a review form. There they can rate the item from 1 to 5 and leave a review. Upon submission, the system will validate the input. If there are any errors, the corresponding fields will be highlighted. If all inputs are valid, system will save new review. After that, the customer will be redirected to the item detail view.

As part of the review functionality, customers should also be able to edit or delete previously submitted reviews.

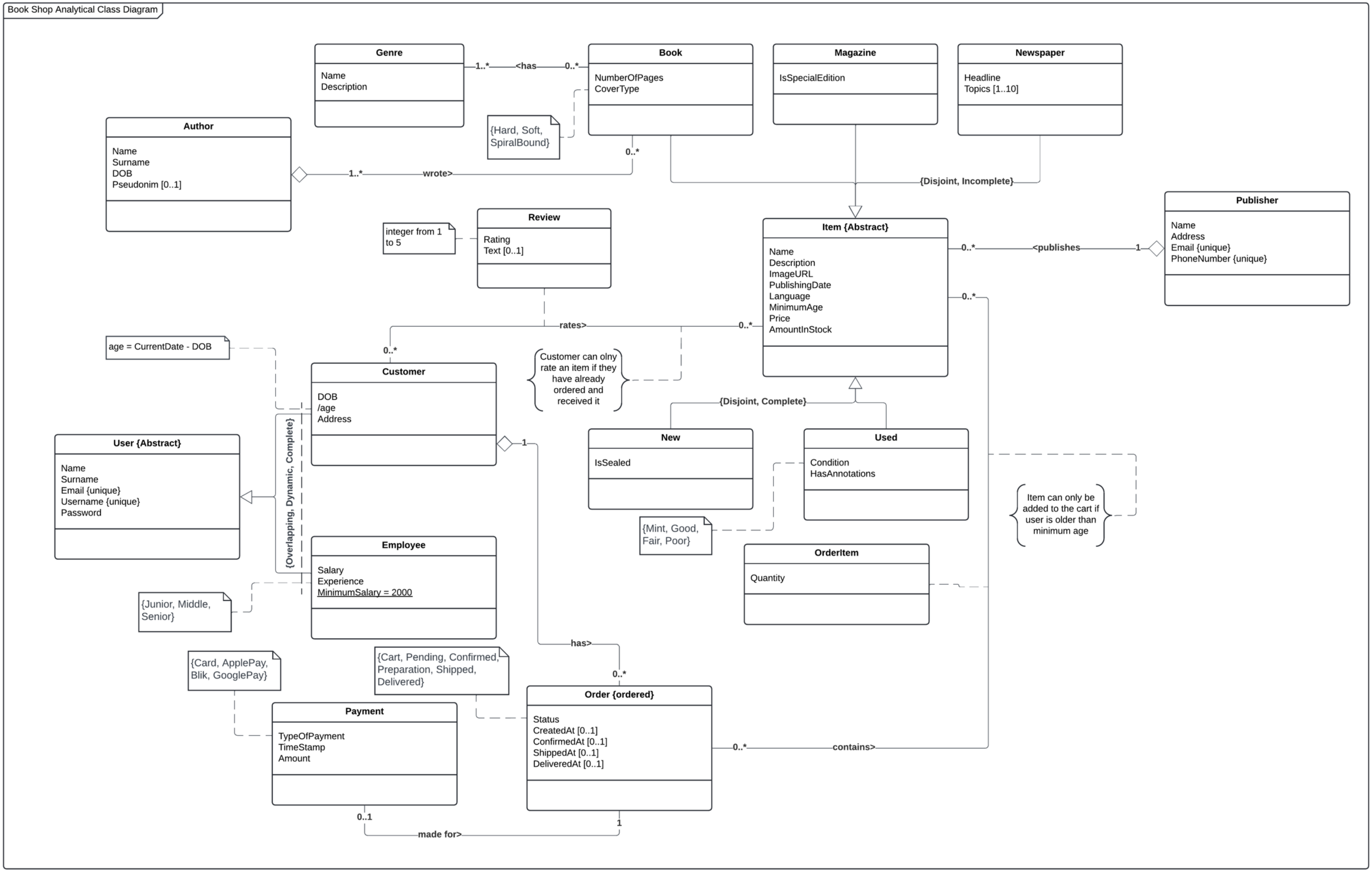
Employees of the store will be able to add new items to the system. They should also be able to browse the catalog with options to edit or delete existing items. Additionally, they will be able to view all confirmed orders that are awaiting fulfillment. For each such order, employees should be able to initiate the preparation process, which includes all steps required to ready the order for shipment. As part of this process, the shipment must be handled, which involves completing the final steps and updating the order status accordingly. When the package will arrive system should be notified by shipping company.

Every day system checks and deletes all carts that were created more than a month ago.

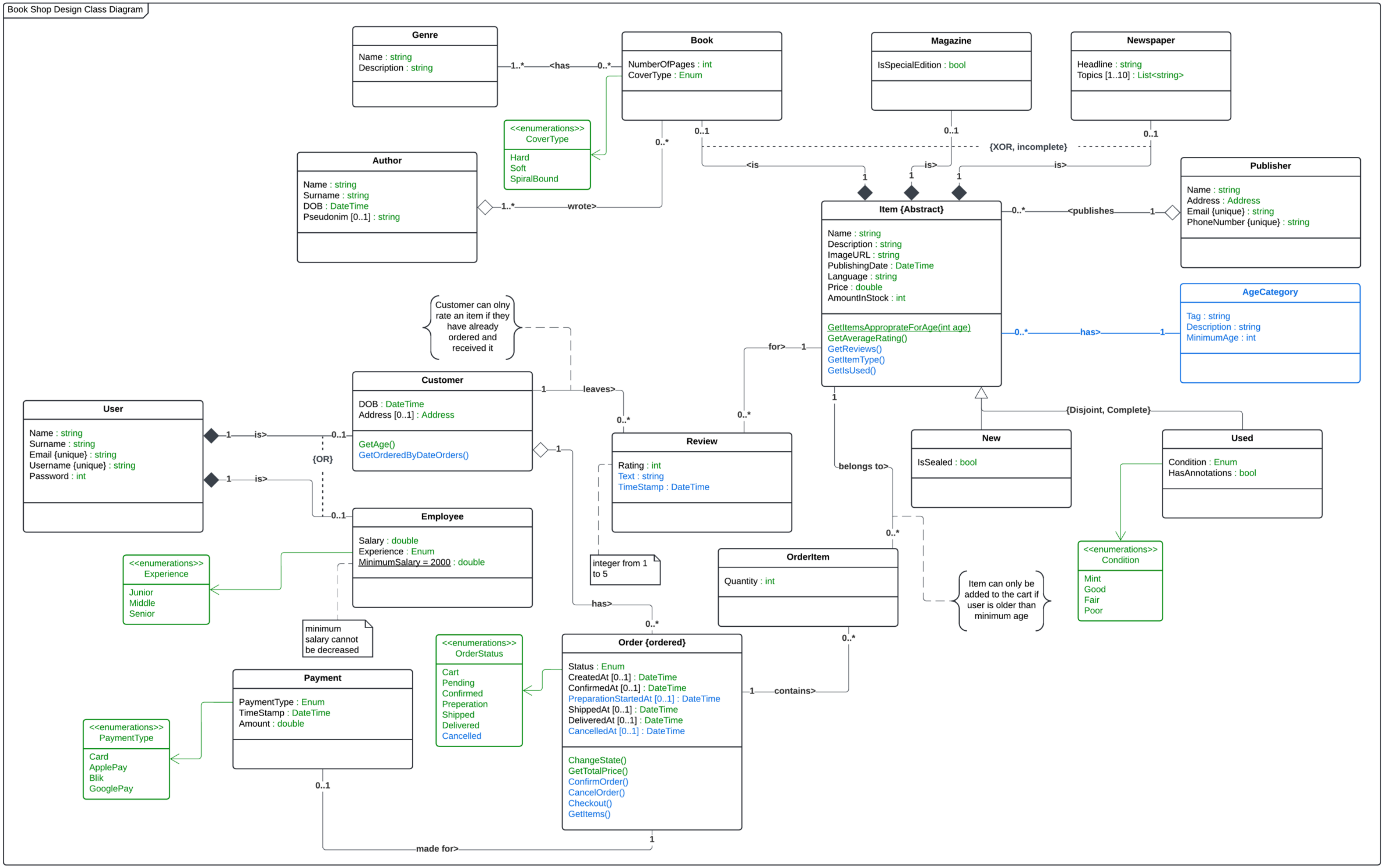
Use Case Diagram



Analytical Diagram



Design Diagram



‘Leave Review’ Use Case Scenario

Actor: Customer

Purpose and context: Customer leaves a review and rates an item

Assumptions: Actor has orders that were delivered; Actor has not left any reviews yet

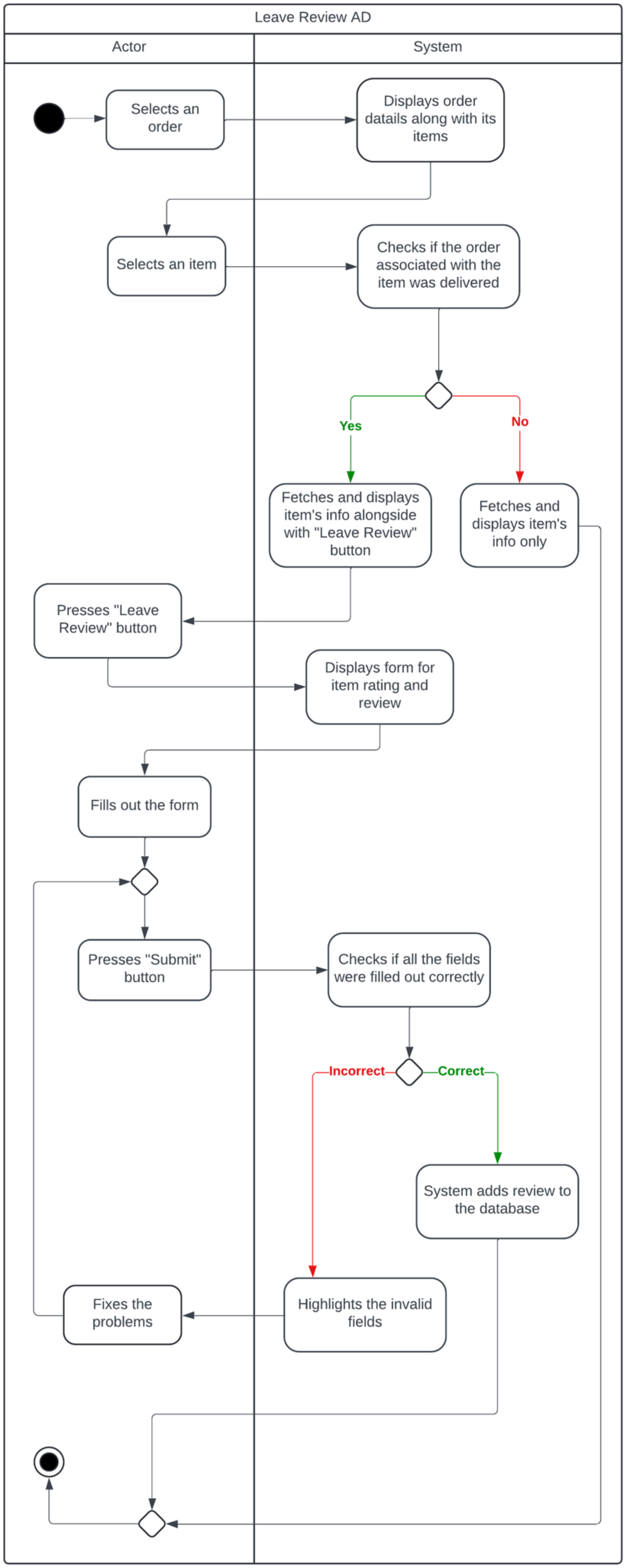
Pre-conditions: Actor sees list of their orders

Basic flow of events:
1. Actor selects one of the orders from the list
2. System displays the order details along with its items
3. Actor selects item they want to leave review for
4. System checks weather the order associated with the item was delivered
5. System fetches and displays item's info alongside with "Leave Review" button
6. Actor clicks on the "Leave Review" button
7. System displays form for item rating and review
8. Actor rates the item and leaves a review
9. Actor clicks "Submit" button
10. System checks if all the fields were filled out correctly
11. System adds newly created item to database

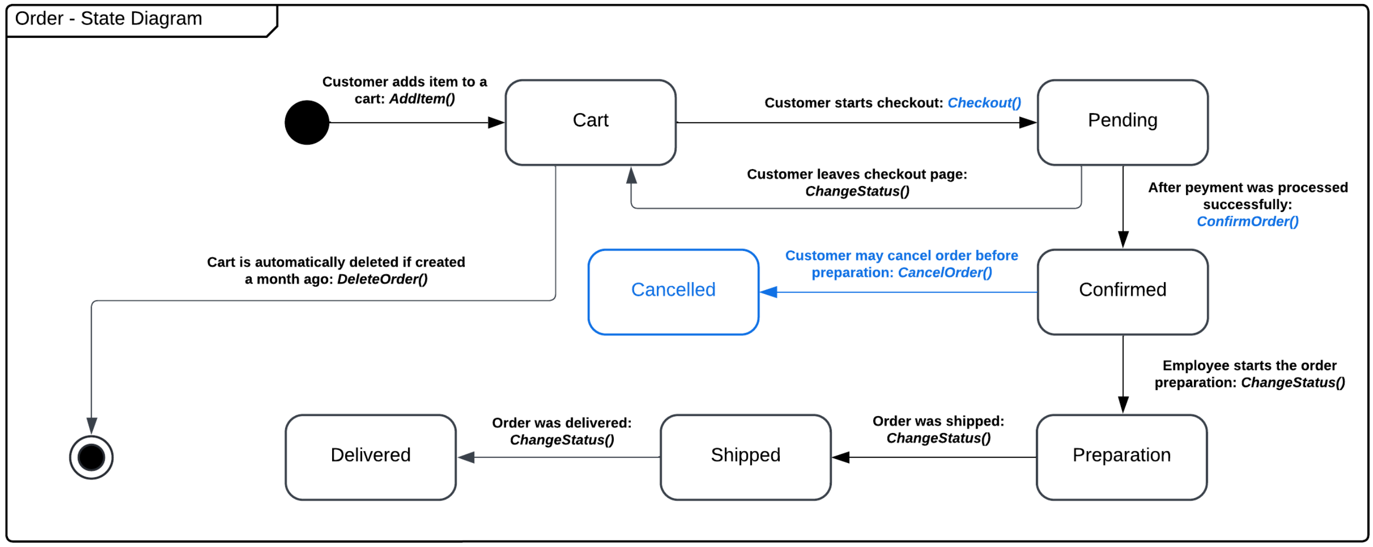
Alternative flow of events:
- **Actor wants to rate an item they haven't received yet**
  5a1. System fetches and displays item’s info only (no button to rate the item)
- **Missing values or invalid inputs were found**
  11a1. System highlights invalid fields in the form
  11a2. Actor fixes problems
  11a3. Actor is back to step 9 (clicks "Submit" button)

Post-condition: Actor is being redirected to the previous view (rated item info)

‘Leave Review’ Activity Diagram


Leave Review AD
Actor
System
Selects an order
Displays order datails along with its items
Selects an item
Checks if the order associated with the item was delivered
Yes
No
Fetches and displays item's info alongside with "Leave Review" button
Fetches and displays item's info only
Presses "Leave Review" button
Displays form for item rating and review
Fills out the form
Presses "Submit" button
Checks if all the fields were filled out correctly
Incorrect
Correct
System adds review to the database
Fixes the problems
Highlights the invalid fields

Order – State Diagram


Order - State Diagram
Customer adds item to a cart: AddItem()
Cart
Customer starts checkout: Checkout()
Pending
Customer leaves checkout page: ChangeStatus()
After peyment was processed successfully: ConfirmOrder()
Cart is automatically deleted if created a month ago: DeleteOrder()
Cancelled
Customer may cancel order before preparation: CancelOrder()
Confirmed
Employee starts the order preparation: ChangeStatus()
Delivered
Order was delivered: ChangeStatus()
Shipped
Order was shipped: ChangeStatus()
Preparation

State Diagram – Design Description

The following state diagram describes possible statuses of an order. *Highlighted changes were described in dynamic analysis.*

Cart – An order is automatically created and assigned the "Cart" status when a customer adds any item to an empty cart. If the cart is not checked out within one month from its creation, it is automatically deleted.

Pending – When the customer initiates the checkout process, the order status changes to "Pending." If customer leaves checkout unfinished status of the order is changed to “Cart” again.

Confirmed – Upon successful completion of the checkout and payment process, the order status is updated to "Confirmed."

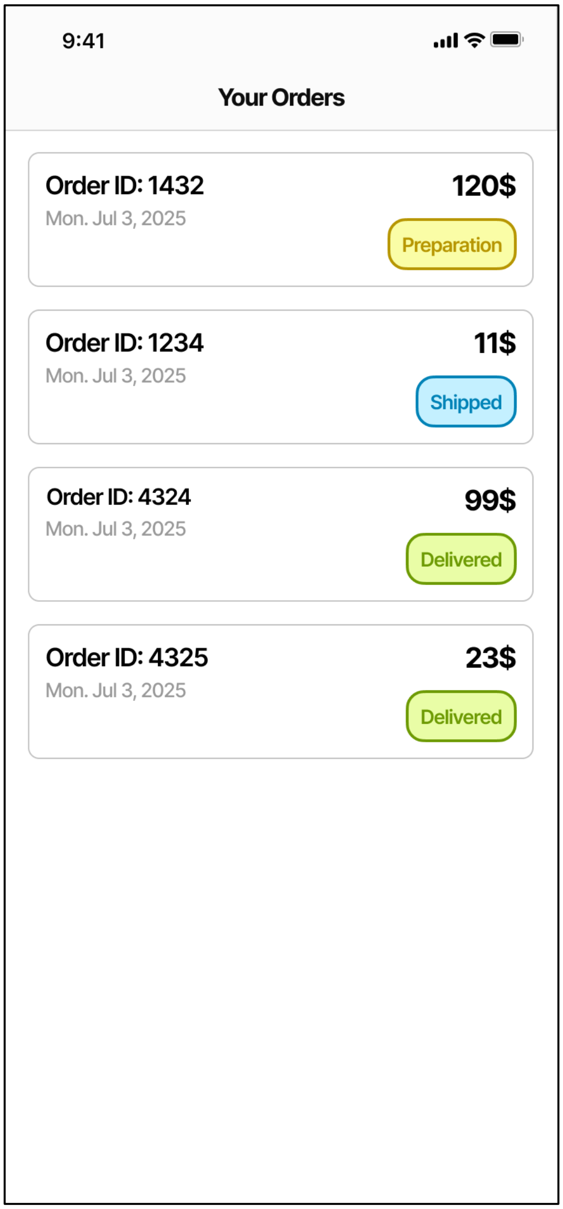
Preparation – Employees can access orders with the "Confirmed" status and begin the preparation process. Once initiated, the status changes to "Preparation."

Shipped – After preparation is complete, the employee dispatches the order and updates its status to "Shipped."

Delivered – When the parcel is successfully collected by the customer, system gets notified by shipping company and status is set to “Delivered”.

‘Leave Review’ GUI Design

‘Your Orders’ View



9:41

Your Orders

Order ID: 1432 120$
Mon. Jul 3, 2025
Preparation

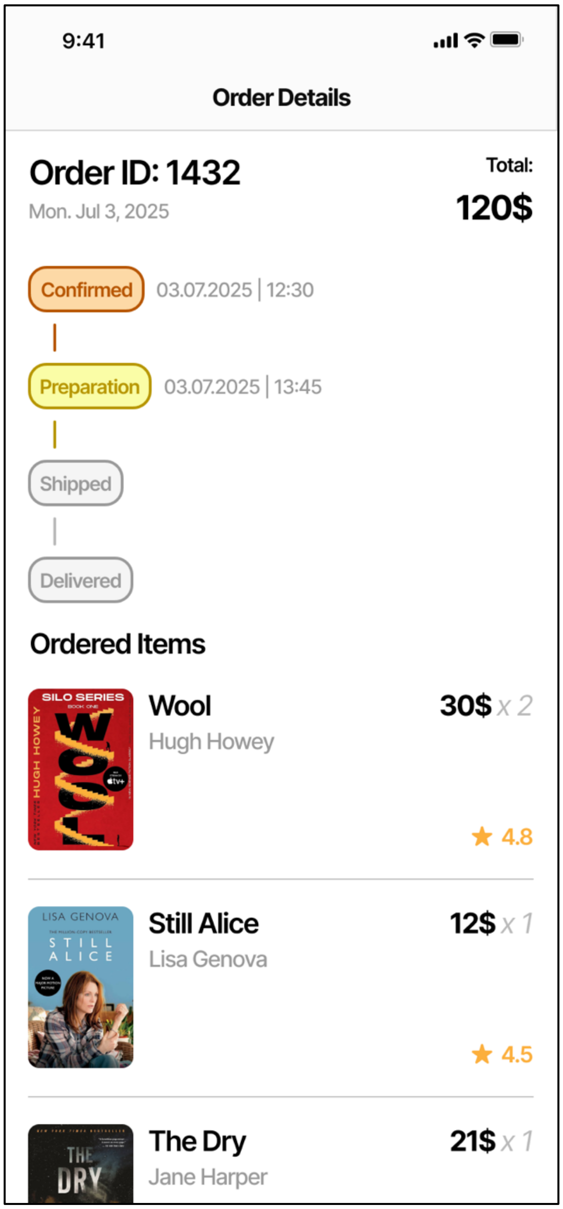
Order ID: 1234 11$
Mon. Jul 3, 2025
Shipped

Order ID: 4324 99$
Mon. Jul 3, 2025
Delivered

Order ID: 4325 23$
Mon. Jul 3, 2025
Delivered

View where customer can browse their orders. Every order is described by its ID, date of last status change, total price and current status. Upon clicking on any of the orders, customer will be directed to the order detail’s view.

‘Order Details’ View



9:41

Order Details

Order ID: 1432

Mon. Jul 3, 2025

Total:

120$

Confirmed 03.07.2025 | 12:30

Preparation 03.07.2025 | 13:45

Shipped

Delivered

Ordered Items

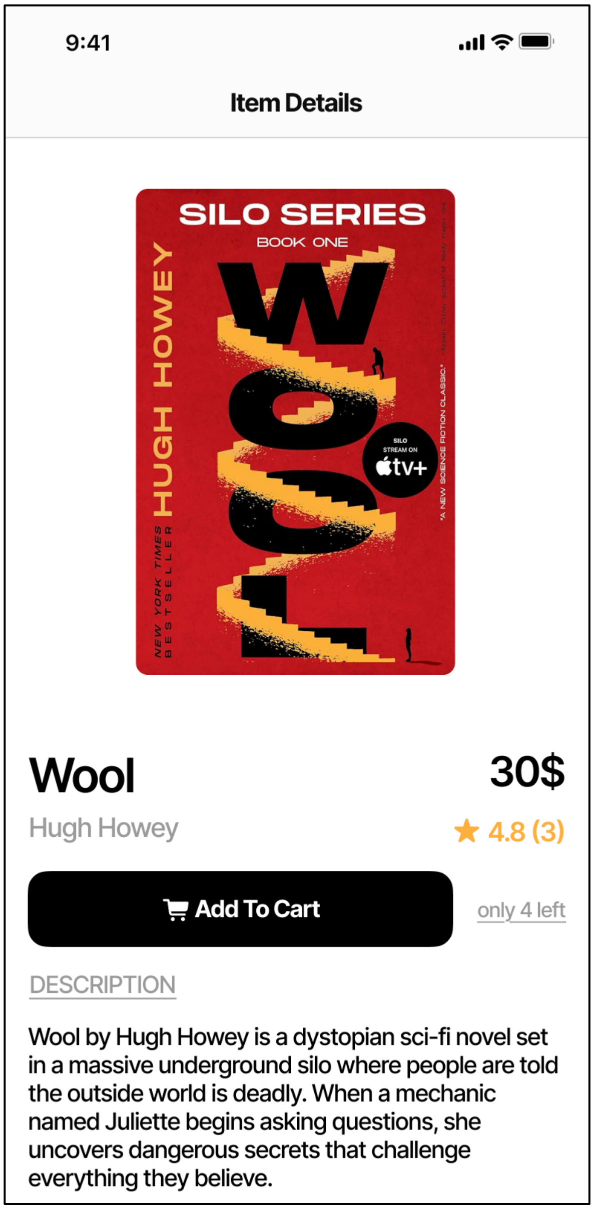
Wool
Hugh Howey
30$ x 2
★ 4.8

Still Alice
Lisa Genova
12$ x 1
★ 4.5

The Dry
Jane Harper
21$ x 1

In order details view we can see the order’s ID, date of last status change, total price. Additionally, there is a graph that shows history of statuses and list of associated items. Upon clicking on any of the items customer will be directed to item detail’s view.

‘Item Details’ View



Item Details

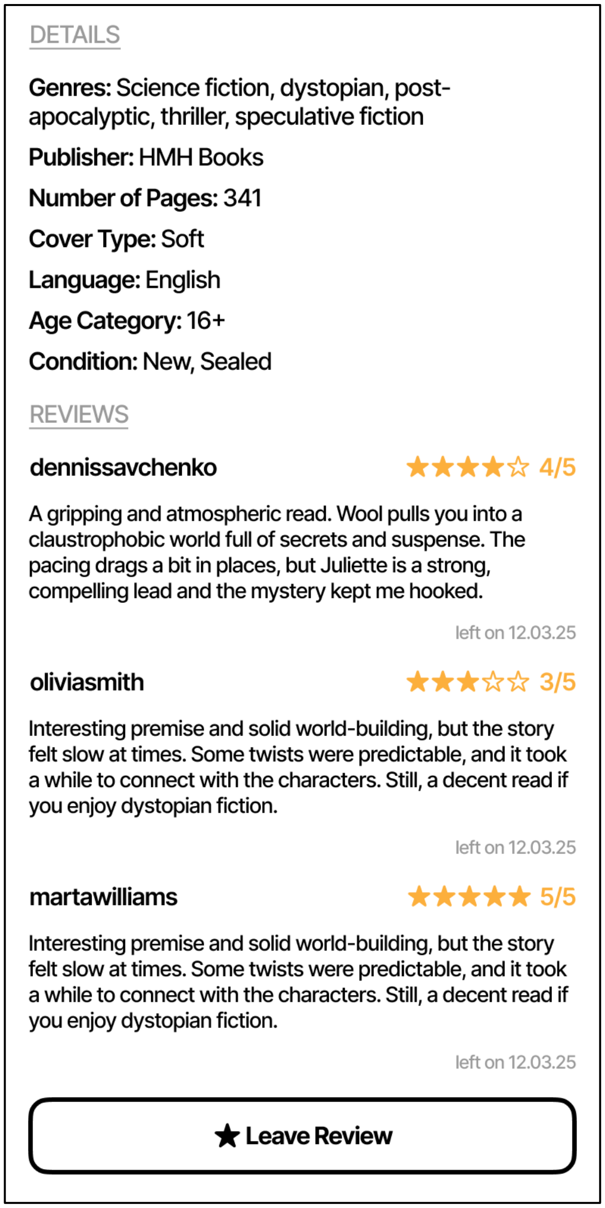
Wool **30$**

Hugh Howey ★ 4.8 (3)

Add To Cart only 4 left

DESCRIPTION

Wool by Hugh Howey is a dystopian sci-fi novel set in a massive underground silo where people are told the outside world is deadly. When a mechanic named Juliette begins asking questions, she uncovers dangerous secrets that challenge everything they believe.



DETAILS

Genres: Science fiction, dystopian, post-apocalyptic, thriller, speculative fiction

Publisher: HMH Books

Number of Pages: 341

Cover Type: Soft

Language: English

Age Category: 16+

Condition: New, Sealed

REVIEWS

dennissavchenko ★★★★☆ 4/5

A gripping and atmospheric read. Wool pulls you into a claustrophobic world full of secrets and suspense. The pacing drags a bit in places, but Juliette is a strong, compelling lead and the mystery kept me hooked.

left on 12.03.25

oliviasmith ★★★☆☆ 3/5

Interesting premise and solid world-building, but the story felt slow at times. Some twists were predictable, and it took a while to connect with the characters. Still, a decent read if you enjoy dystopian fiction.

left on 12.03.25

martawilliams ★★★★★ 5/5

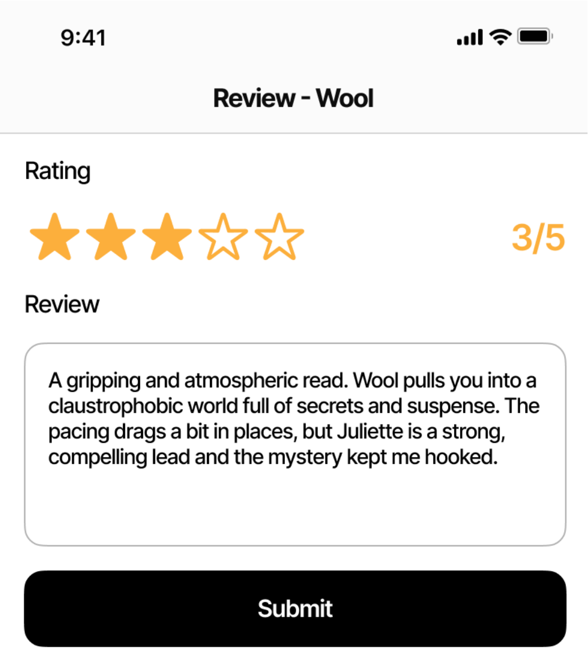
Interesting premise and solid world-building, but the story felt slow at times. Some twists were predictable, and it took a while to connect with the characters. Still, a decent read if you enjoy dystopian fiction.

left on 12.03.25

★ Leave Review

In the item details view, the name, author, price, image, and rating are displayed. Below this header information, there is an "Add to Cart" button. Under the button, the item's description is shown. This is followed by a section with detailed item information and a review section. If the customer has received the item, a "Leave Review" button will appear. Otherwise, nothing will be displayed after review section.

‘New Review’ View



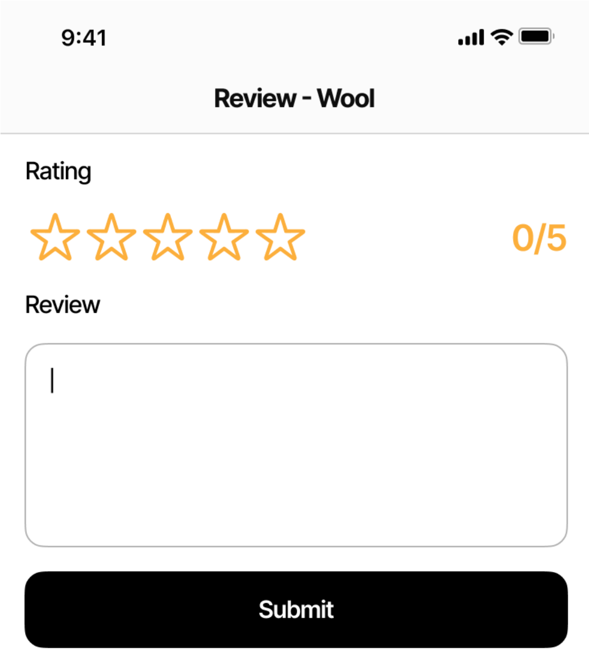
Review - Wool

Rating

★★★☆☆ 3/5

Review

A gripping and atmospheric read. Wool pulls you into a claustrophobic world full of secrets and suspense. The pacing drags a bit in places, but Juliette is a strong, compelling lead and the mystery kept me hooked.

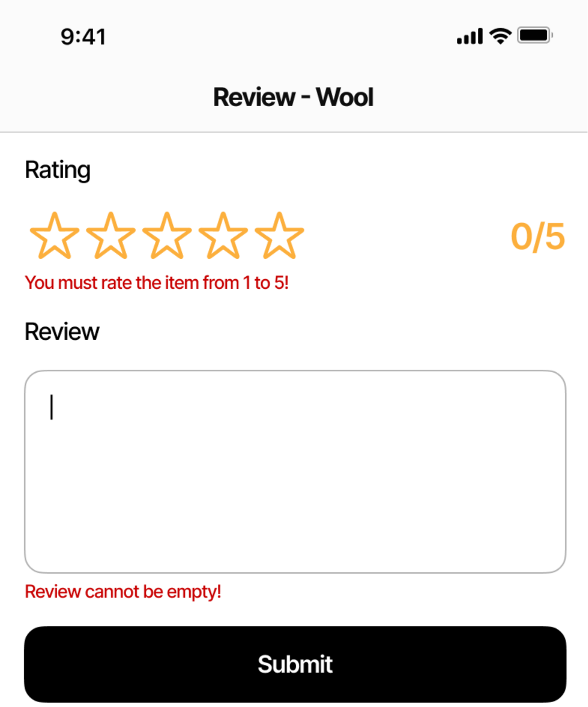


Review - Wool

Rating

☆☆☆☆☆ 0/5

Review



Review - Wool

Rating

☆☆☆☆☆ 0/5

You must rate the item from 1 to 5!

Review

Review cannot be empty!

This view contains rating component, edit text field and the ‘Submit’ button. Upon clicking on the button, the field will we verified. If everything is filled out correctly customer will be redirected to the item detail’s view, otherwise corresponding error messages will appear.

Discussion of Design Decisions

The class diagram illustrates the design tailored for C# implementation. It stores all the object’s information in a database. We utilize Entity Framework code first approach to create the database structure and efficiently manipulate data using LINQ queries. For database implementation we have chosen file-based SQLite as local hosting was acceptable according to the project rules.

Attributes Validation

In a C# Web API, it is considered good practice to implement validation on both the API and database levels. This double-sided validation ensures data integrity and prevents invalid modifications, including those made directly via SQL.

At the database level, validation is enforced using custom constraints along with Entity Framework's Fluent API methods. At the API level, validation is applied to DTOs (Data Transfer Objects) through class-based and custom data annotations. These annotations help ensure the correctness of input data when creating or updating records in the database.

A special case in validation is checking for uniqueness. At the database level, this can be enforced using a unique constraint defined via the Entity Framework Fluent API. However, API-level uniqueness validation must be handled in the server layer. This involves querying the database to verify whether the value has already been used before proceeding with the operation.

Optional Attributes

In C#, optional attributes are typically implemented through nullable properties or by providing default values. They are commonly used in DTOs, request models, and entity classes to represent fields that may or may not be supplied by the client or data source.

Multi-Value Attributes

For multi-value attributes, the system uses standard C# collections such as *ICollection<T>*, which are stored in the database as JSON strings within a single column. This is achieved using a value converter in Entity Framework that serializes the list to JSON when saving data and deserializes it back to a collection when reading. This approach allows storing and retrieving lists without creating additional tables or relationships.

Complex Attributes

For complex attributes represented by custom classes, the system uses nested objects called owned types. These do not have their own identity and are stored in the same table as the owning entity, with column names prefixed by the property name. Entity Framework maps them using *OwnsOne()* in the *OnModelCreating* method, allowing the data to be saved and loaded as part of the parent entity.

Derived Attributes

Derived attributes are implemented using C#'s get property feature. These attributes are automatically calculated based on other properties and do not require additional storage. For example, a customer’s age is derived from their date of birth.

Derived attributes are recalculated dynamically and are not explicitly stored in database, ensuring they are always up to date.

Tagged Values

Tagged values are implemented using enumeration classes. Enumerations provide a strongly typed representation for attributes with predefined sets of values, ensuring clarity and consistency. Enums are stored in the database as their underlying numeric values by default, using *HasConversion<int>()* to map the enum to an integer for efficient and compact storage.

Data Persistency

The system uses Entity Framework Core with an SQLite database to manage persistent storage of application data. This solution ensures reliable, scalable, and structured data management without requiring external database servers.

All domain entities are represented by C# classes mapped to database tables using Entity Framework. Each entity includes a primary key (typically an Id property), ensuring unique identification. The *DbContext* class defines the database structure, including tables via *DbSet<T>* properties, and manages connections, queries, and transactions. Data is stored in a local SQLite file, making it lightweight and easy to distribute. EF Core handles the creation and update of the database schema via migrations or automatic model syncing.

Create, Read, Update, Delete operations are performed using EF Core's LINQ-based API. Changes are tracked automatically, and calling *SaveChanges()* commits them to the SQLite database.

Data is queried through LINQ expressions, allowing filtering, projection, and eager/lazy loading of related entities. Asynchronous operations, async and await, ensure responsive data access.

Associations

Entity Framework handles associations between classes using navigation properties and relational mapping rules. There are three main types of associations: one-to-one, one-to-many, and many-to-many. Each type is modeled in C# using object references and collections and configured either by convention or explicitly using the Fluent API.

Basic Relationships

In a one-to-many relationship, one entity relates to many others. This is set up using *HasMany().WithOne(),* where the "many" side holds the foreign key. EF automatically maps the relationship and keeps the link consistent.

In a many-to-many relationship, both entities can have multiple related entries. It’s configured with *HasMany().WithMany()* and *UsingEntity()* to define the join table with two foreign keys as a composite key. No extra class is needed unless additional data is stored in the link.

In a one-to-one relationship, each entity is linked to just one other. EF maps this using *HasOne().WithOne(),* with one side holding a foreign key or sharing a primary key. The relationship is enforced to be unique on both sides.

Association with an Attribute

To implement an association with an attribute, we need to create a separate entity to represent it. This entity will store references to the two associated classes and the additional attributes. Each association is modeled as a basic one-to-many relationship. Additionally, cascade delete is applied so that when one of the related entities is removed, the corresponding attribute entity is also deleted.

Multi-Aspect

The Item entity has a multi-aspect structure. The first aspect distinguishes between New and Used items (disjoint and complete). This is implemented as regular inheritance at the server level, allowing access to Item, NewItem, and UsedItem separately through individual DbSets. However, in the database, all these classes are stored in a single table using a discriminator column.

The second aspect includes *Book, Magazine, and Newspaper*, and is defined as disjoint and incomplete. In the design, this is implemented as a composition, since this aspect introduces additional attributes, and Book also has two associations with other classes. Each subtype is modeled as a separate class and linked to Item through a one-to-one composition association. An *{XOR, incomplete}* constraint is applied to the group of associations, meaning that each Item can be associated with only one or none of these types at a time. Cascade delete is used to ensure that when an Item is removed, its associated object (if any) is also deleted. Since Item is not an abstract class, it does not require a mandatory link to one of the aspect classes.

Overlapping, Dynamic Inheritance

Another inheritance in the system differentiates *User into Customer and Employee*. This was also implemented as a composition, since Customer has its own associations with other classes. The User entity has a one-to-one relationship with both Customer and Employee, with an *{OR}* constraint indicating that each user must be associated with at least one of these subtypes. Cascade delete is applied to these associations to ensure that when a User is deleted, the related Customer or Employee object is also removed.

Business Constraints

Two custom business constraints are enforced in the system. Customers cannot order a book unless they meet its minimum age requirement, and they cannot submit a review for an item that has not been delivered. These rules are checked on the server side, and violations result in an exception being thrown.

Ordered Classes

Classes marked as *{ordered}* have a meaningful sequence among their instances. Although they are stored in the database without enforcing order at the schema level, ordering is applied during data retrieval using appropriate sorting.

Dynamic Analysis

Based on the previously presented diagrams, a dynamic analysis was conducted. This analysis showed the need for further expansion of the design and state diagrams. We had to incorporate newly identified system requirements and behaviors. *The implemented changes were highlighted in blue on the diagrams.*

State diagram - Order

After careful consideration, we have decided to add special methods to change some of the order's statuses.

Checkout() – method that finalizes the cart, collects all necessary data of the user and the cart for the order to be complete.

ConfirmOrder() – was added as a separate method as it also includes creating and verifying a payment, saving it to the database.

CancelOrder() – is a specific method that allows the user to cancel the order, but only before preparation has started. Additionally, *CancelledAt* optional attribute was added.

Order

GetItems() – this method was added to fetch items associated with the order along with their quantities, in order to display them in the order details.

Item

During the GUI design phase, it was decided that removing the *MinimumAge* attribute would be a better solution, as there are generally only a few age categories that items are associated with. Therefore, an *AgeCategory* class was created with *Tag*, *Description*, and *MinimumAge* attributes.

Also, we remembered that the item view needs a list of reviews, not just the average rating. Hence, the *GetReviews()* method was added.

GetItemType() – method that return item type as book, magazine, newspaper or without type (null).

GetIsUsed() – needed to determine whether item inherits from used or new item.

Review

In Review class's *Text* attribute went from nullable to required and *TimeStamp* was added to save the date of the review posting.

Customer

GetOrdersOrderedByDate() – allows the customer to retrieve their orders sorted by the most recent update date, with the latest orders appearing at the top.