Software Architecture & Design Assignment #1

Software Design using a UML Class Diagram

Assignment#1 SW Design-Pair Share: 5

GioMonci, Joseph Lennon

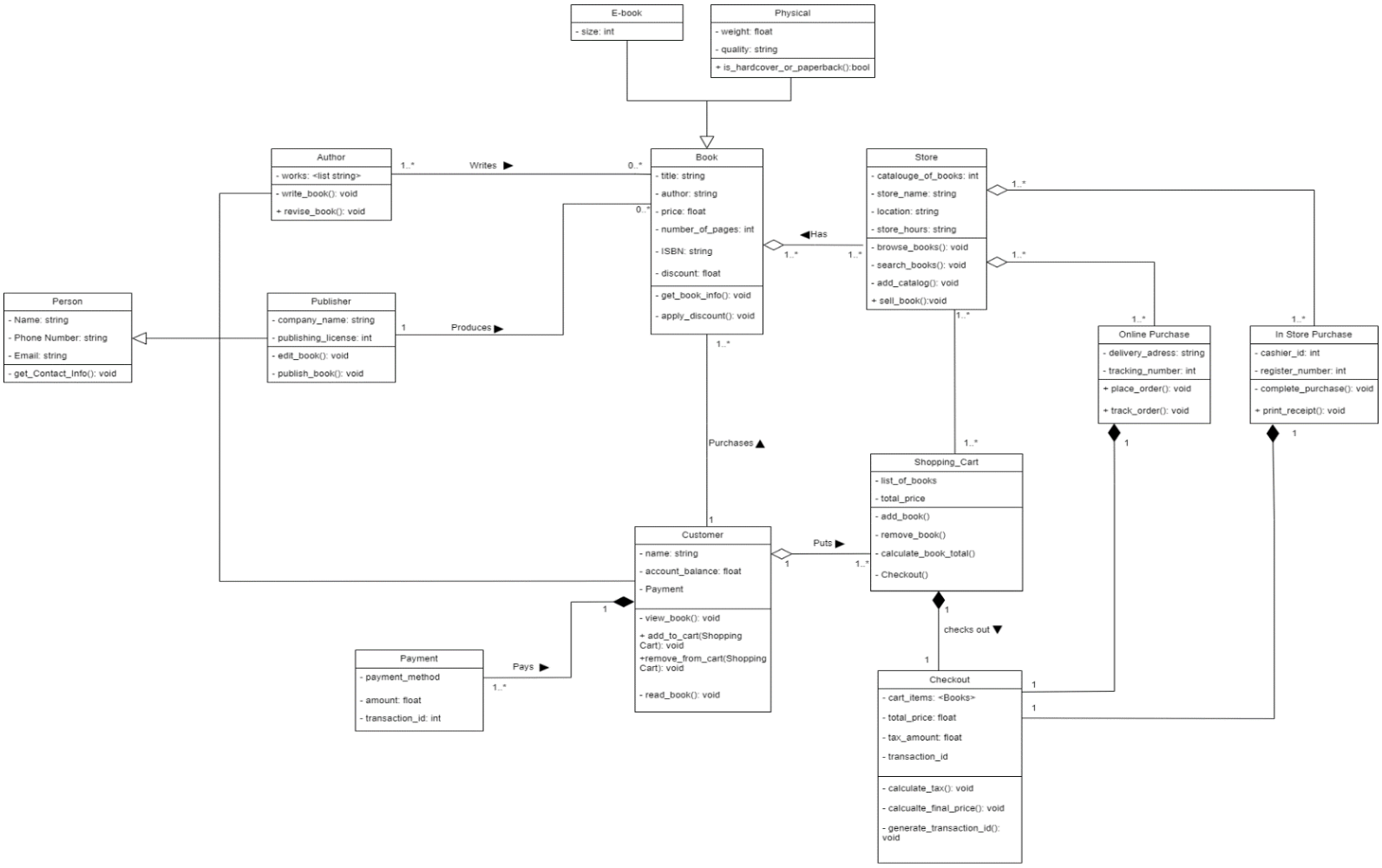
Department of Engineering, Florida Gulf Coast University

CEN 4065 - Software Architecture & Design

Dr. Buckley

Software Architecture & Design Assignment #1

1) Question 1: A designer has been assigned a task to model a book subsystem with a class diagram. Knowing that an author writes a book, then a publisher produces it, and a customer purchases it, **improve the given class diagram** to model the subsystem, showing class attributes, member functions, relationships, associations, roles, and cardinality, descriptions on connecting lines, and correct UML syntax. Provide a full explanation that describes your class diagram as well (10 points).



Description of Class Diagram:

Person: represents general information about a person, such as their name, phone number, and email. Base class for more specific classes: Author, Publisher, Customer.

Relations:

- Customer class: Relation: Inheritance (generalization); Multiplicity: 1 to 1
- Author class: Relation: Inheritance (generalization); Multiplicity: 1 to 1
- Publisher class: Relation: Inheritance (generalization); Multiplicity: 1 to 1

Author: represents authors who write books. Manages author details and books they have written.

Relations:

- Person class: Relation: Inheritance (generalization); Multiplicity: 1 to 1
- Book class: Relation: Association; Multiplicity: 1 to many

Publisher: represents publishers responsible for producing books. It manages the process of editing, publishing, and distributing books.

Relations:

- Person class: Relation: Inheritance (generalization); Multiplicity: 1 to 1
- Book class: Relation: Association; Multiplicity: 1 to many

Customer: represents people who purchase books from the store. It manages customer details and purchasing actions.

Relations:

- Book class: Relation: Association; Multiplicity: 1 to many
- Shopping Cart class: Relation: Association; Multiplicity: 1 to 1
- Payment class: Relation: Association; Multiplicity: 1 to 1
- Person class: Relation: Inheritance (generalization); Multiplicity: 1 to 1

Payment: manages the payment details for purchases, whether online or in-store.

Relations:

- Customer class: Relation: Composition; Multiplicity: 1...* to 1

Shopping Cart: holds items that a customer intends to purchase, whether online or for in-store pickup.

Relations:

- Customer class: Relation: Aggregation; Multiplicity: 1 to Many
- Store class: Relation: Association; Multiplicity: Many to Many
- Shopping Cart class: Relation: Composition; Multiplicity: 1 to 1

Book: represents books that the store sells. It holds attributes like title, price, number of pages, quantity, and author.

Relations:

- Store class: Relation: Aggregation; Multiplicity: many to 1
- Author class: Relation: Association; Multiplicity: 1 to many
- Publisher class: Relation: Association; Multiplicity: 1 to many
- Customer class: Relation: Association; Multiplicity: 1 to many
- E-book class: Relation: Inheritance (generalization); Multiplicity: 1 to 1
- Physical Book (Paperback) class: Relation: Inheritance (generalization); Multiplicity: 1 to 1

E-Book: represents

Relations: specialized version of the Book class and represents digital books

- Book class: Relation: Inheritance (generalization); Multiplicity: 1 to 1

Physical: specialized version of the Book class, representing physical books with physical attributes.

Relations:

- Book class: Relation: Inheritance (generalization); Multiplicity: 1 to 1

Store: represents the store in which you can go in person or online to purchase books. It manages relationships with books, purchases (online and in-store), and carts. It enables book sales and tracks inventory.

Relations:

- Book class: Relation: Aggregation; Multiplicity: many to many
- Online Purchase class: Relation: Aggregation; Multiplicity: many to many
- In-Store Purchase class: Relation: Aggregation; Multiplicity: many to many
- Shopping Cart class: Relation: Association; Multiplicity: many to many

Online Purchase: handles purchases made online. It includes information on the purchase, payment method, and items bought.

Relations:

- Store class: Relation: Aggregation; Multiplicity: 0..1 to many
- Payment class: Relation: Composition; Multiplicity: 1 to 1

In Store Purchase: manages purchases made in a physical store, tracking payment and items bought.

Relations:

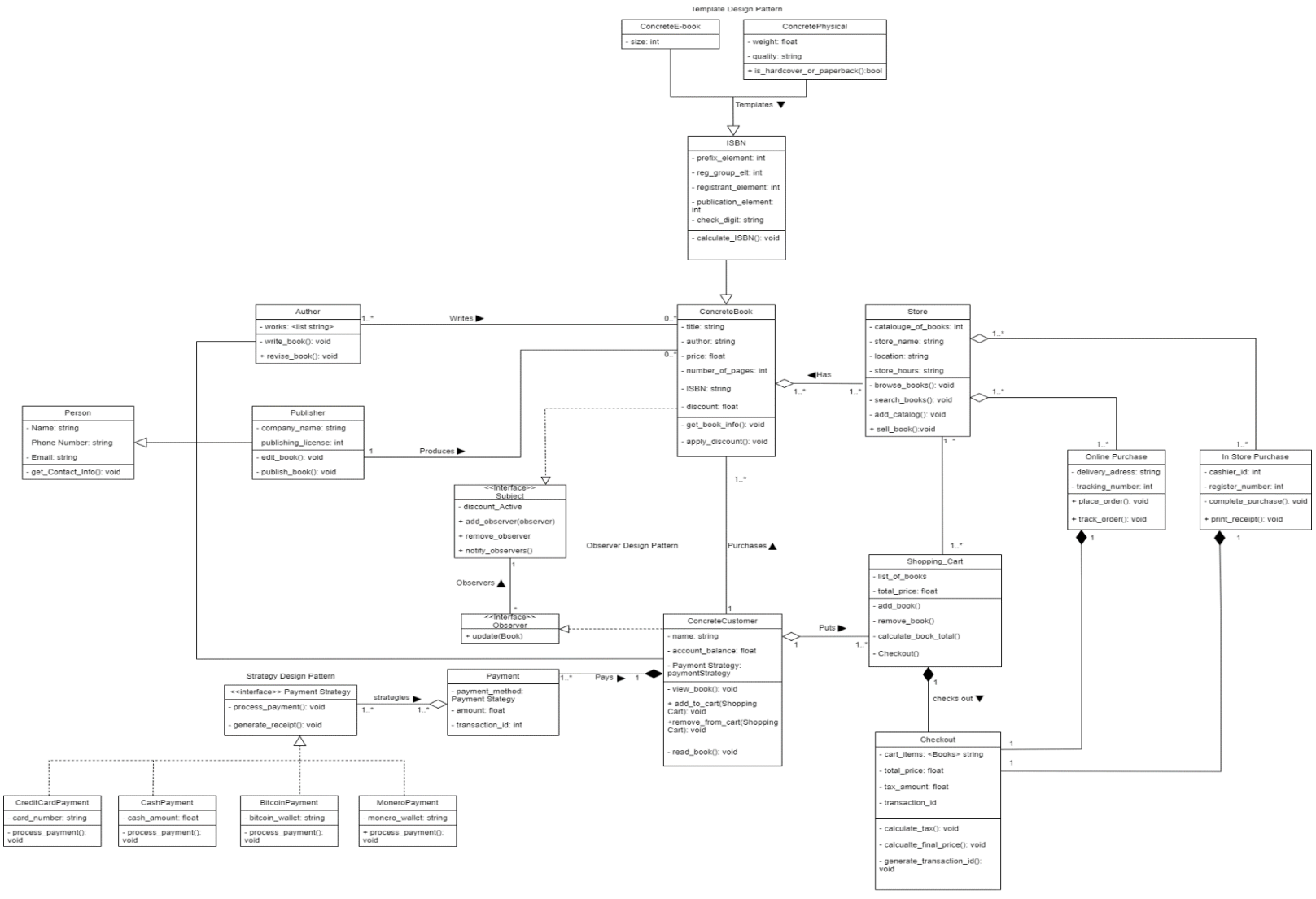
- Store class: Relation: Aggregation; Multiplicity: 0..1 to many
- Payment class: Relation: Composition; Multiplicity: 1 to 1

Checkout: represents the checkout process in store and online. Adds together the cart item price, discounts, and tax amount to give a final price.

Relations:

- Shopping Cart class: Relation: Composition; Multiplicity: 1 to 1
- Online Purchase class: Relation: Composition; Multiplicity: 1 to 1
- In Store Purchase class: Relation: Composition; Multiplicity: 1 to 1

2) Question 2: Extend the class diagram you updated in question#1 by adding a design pattern to it. Select an appropriate design pattern to solve an **essential** problem and improve the system design. You must justify and explain why you have chosen a given design pattern, the exact problem it is solving in a given context, and why the solution to that problem is essential. Your extended diagram must follow all the syntax of the UML class diagram (10 points)



Observer Design Pattern:

Problem solved: The system needs to notify multiple customers when certain events occur, such as a discount becoming active on a book. The problem here is how to efficiently notify all interested customers of such changes without tight coupling between classes.

Solution: The Observer Pattern allows the Book (the subject) to keep track of all Customer observers (observer). When a discount becomes active, the book can notify all registered customers. This pattern decouples the Book class from the Customer class, as the book doesn't need to know which customers are observing it—it simply notifies all registered observers (observer).

Template Design Pattern:

Problem solved: Both e-books and physical books need to generate an ISBN number, which follows a common algorithm. However, they have distinct characteristics (e.g., size for e-books and weight for physical books) and may require some differences in how they manage their details beyond generating the ISBN. The challenge is how to allow them to share this common ISBN generation logic without duplicating code.

Solution: The Template Design Pattern allows you to define the common steps for generating an ISBN number in a base class (e.g., Book) and let the subclasses, ConcreteE-book and ConcretePhysical, implement specific details related to their unique characteristics (Template). The shared method for generating an ISBN is reused in both book types, but each subclass can extend other behaviors independently (Template).

Strategy Design Pattern:

Problem solved: The bookstore has payment methods such as credit card and cash. However, companies nowadays are accepting more forms of currency such as crypto currency. If we keep adding more payment methods in the payment class, then it will get bloated. This is because each payment method has its own process on how it handles transactions, generating receipts, etc. The challenge is how to allow for different payment methods such as bitcoin and Monero without creating a rigid and bloated class that would be difficult to add functionality too in the future when we inevitably add more payment methods. To solve this problem, we chose the Strategy Design Pattern.

Solution: The Strategy Design Pattern is a behavioral design pattern that defines a family of algorithms, puts them in separate classes, and makes them objects that are interchangeable (strategy). The Strategy Design Pattern is applied by creating an interface (Payment Strategy) that defines process_payment() and generate_receipt(), which are behaviors shared by the payment method classes that inherit the payment strategy (strategy). Each payment method class – CreditCardPayment, CashPayment, BitcoinPayment, or others – applies the interfaces in its own way(strategy). This dynamic design allows the system to switch between different payment methods, allowing for more flexibility to modify payments options without affecting the system or a single class (strategy). For example, if the store wanted to add PayPal in the future, we could simply create a new PayPalPayment class without having to modify the existing code (strategy).

References

"Template Method." *Refactoring.Guru*, refactoring.guru/design-patterns/template-method. Accessed 19 Sept. 2024.

"Observer." *Refactoring.Guru*, refactoring.guru/design-patterns/observer. Accessed 19 Sept. 2024.

"Strategy." *Refactoring.Guru*, refactoring.guru/design-patterns/strategy

"Template Method." *Refactoring.Guru*, refactoring.guru/design-patterns/template-method. Accessed 19 Sept. 2024.

UML Class Diagram Tutorial, www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/