CSCI-C 322 Object-Oriented Software Methods

Assignment1

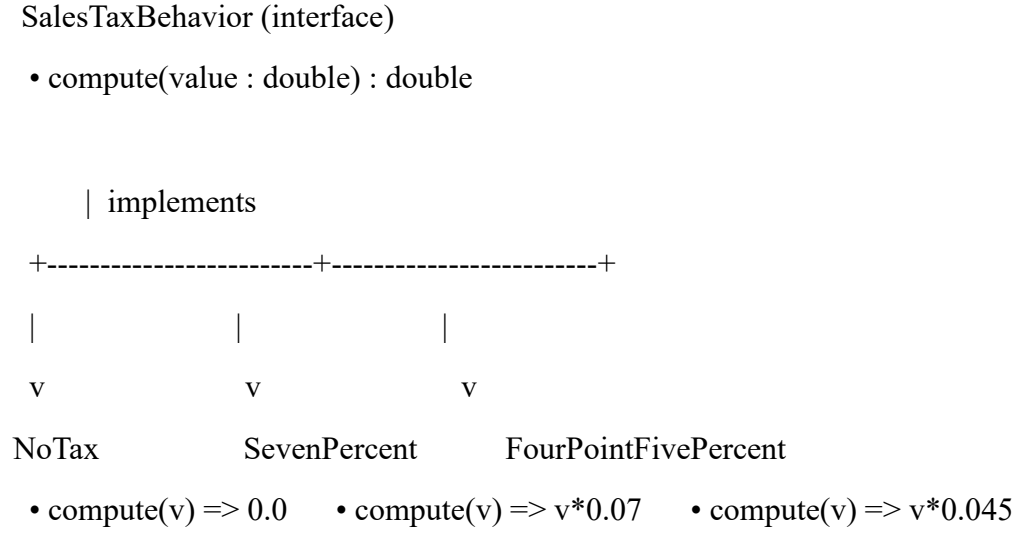
Due by Monday 2/3/2025

Problem 1: Setting Behavior Dynamically

Part A: Continue your work on Lab 02 and perform the following tasks:

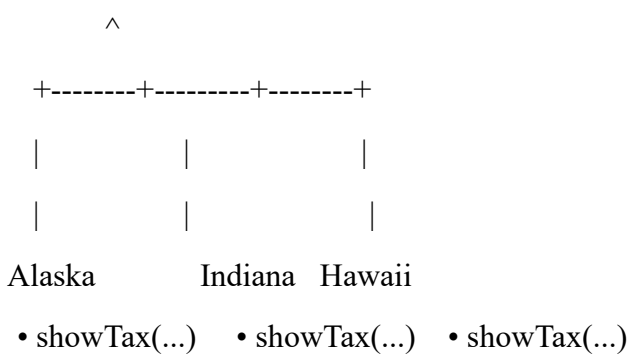
- Add a new sales tax behavior, that is 4.5% of the sale
- Add a new state to the system, that is Hawaii
- Modify the application to allow setting the behavior dynamically
- Modify the application to set the following sales tax behaviors dynamically once the state is identified (Not part of the state constructor):
    - Hawaii (4.5%)
    - Indiana (7%)
    - Alaska (0%)
- Similar to Lab 2, the application should process the command line arguments including Hawaii.

Part B: Visualize the new system design using a UML class diagram.



State (abstract)

- name : String
- salesTaxBehavior : SalesTaxBehavior
- setName(n : String)
- getName() : String
- setSalesTaxBehavior(stb : SalesTaxBehavior)
- showTax(value : double) : abstract



SalesTaxCalculator (class)

- main(args : String[])

  1) Read stateName & saleAmount

  2) Create appropriate State object (Alaska, Indiana, Hawaii)

  3) Dynamically set the right SalesTaxBehavior (NoTax, SevenPercent, or FourPointFivePercent)

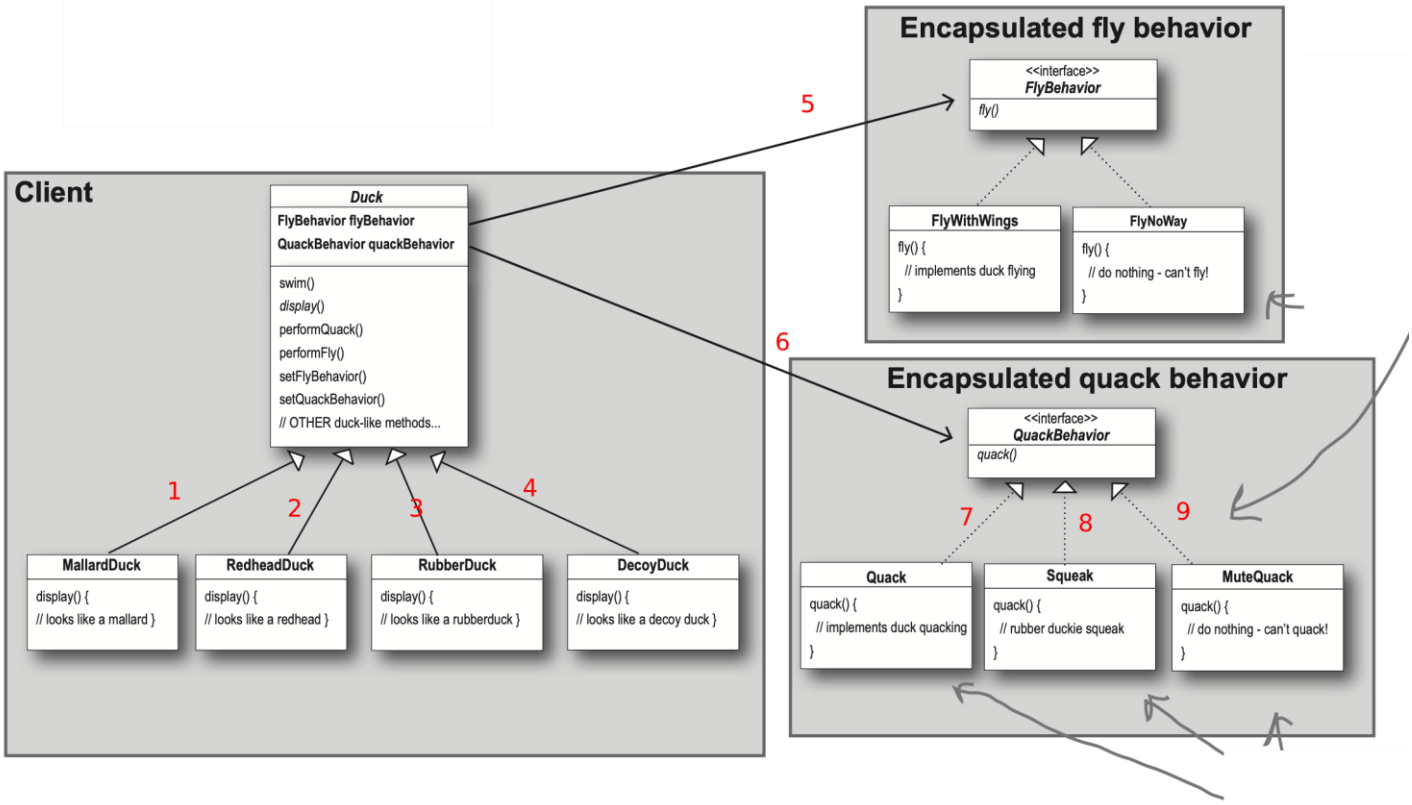
  4) Call showTax(saleAmount) to display result

Problem 2: HAS-A and IS-A

Based on the different relationships in the duck simulator app, categorize the numbered relationships in the

UML class diagram below into HAS-A or IS-A. Justify your classification using one sentence for each one. Here is a **quick reference** to each numbered arrow in the diagram, along with whether it is an **IS-A** or **HAS-A** relationship and a **one-sentence** justification:

1. **MallardDuck → Duck**
    - **IS-A**: MallardDuck extends Duck, so MallardDuck *is* a specialized kind of Duck.
2. **RedheadDuck → Duck**
    - **IS-A**: RedheadDuck extends Duck, indicating RedheadDuck *is* a type of Duck.
3. **RubberDuck → Duck**
    - **IS-A**: RubberDuck extends Duck, so RubberDuck *is* another variety of Duck.
4. **DecoyDuck → Duck**
    - **IS-A**: DecoyDuck extends Duck, therefore DecoyDuck *is* a form of Duck.
5. **Duck → FlyBehavior**
    - **HAS-A**: Duck holds a reference to a FlyBehavior object, so each Duck *has* a FlyBehavior.
6. **Duck → QuackBehavior**
    - **HAS-A**: Duck also maintains a QuackBehavior object, so each Duck *has* a QuackBehavior.
7. **Quack → QuackBehavior**
    - **IS-A**: Quack implements QuackBehavior, meaning Quack *is* a type of QuackBehavior.
8. **Squeak → QuackBehavior**
    - **IS-A**: Squeak implements QuackBehavior, so Squeak *is* a QuackBehavior strategy.
9. **MuteQuack → QuackBehavior**
    - **IS-A**: MuteQuack implements QuackBehavior, indicating it *is* a particular QuackBehavior variant.



Please ignore the hand drawn arrows. They are part of the diagram in the textbook.

Problem 3: The Observer Design Pattern

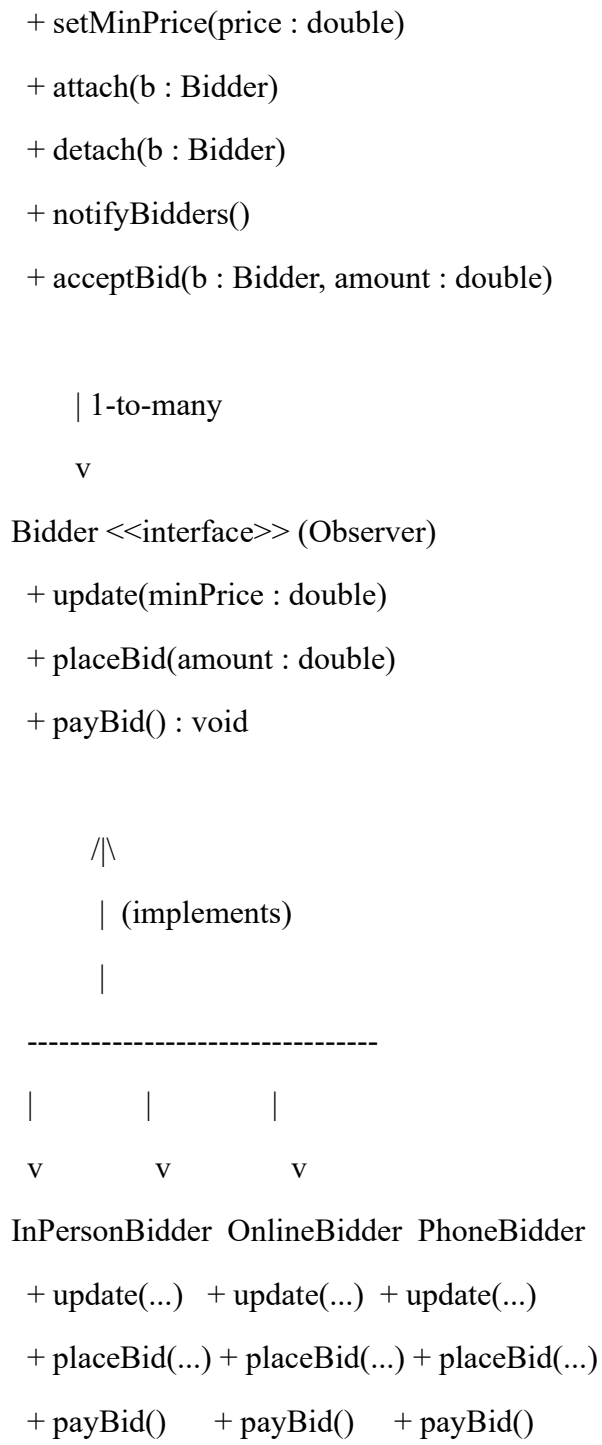
Create a UML class diagram for an auction system. In this system, the Auctioneer sets the minimum bidding price and broadcasts this information to the bidders. In return, the Bidders submit their bids through different channels: in-person, online, or by phone. Therefore, there are three types of bidders, each corresponding to a different bidding channel.

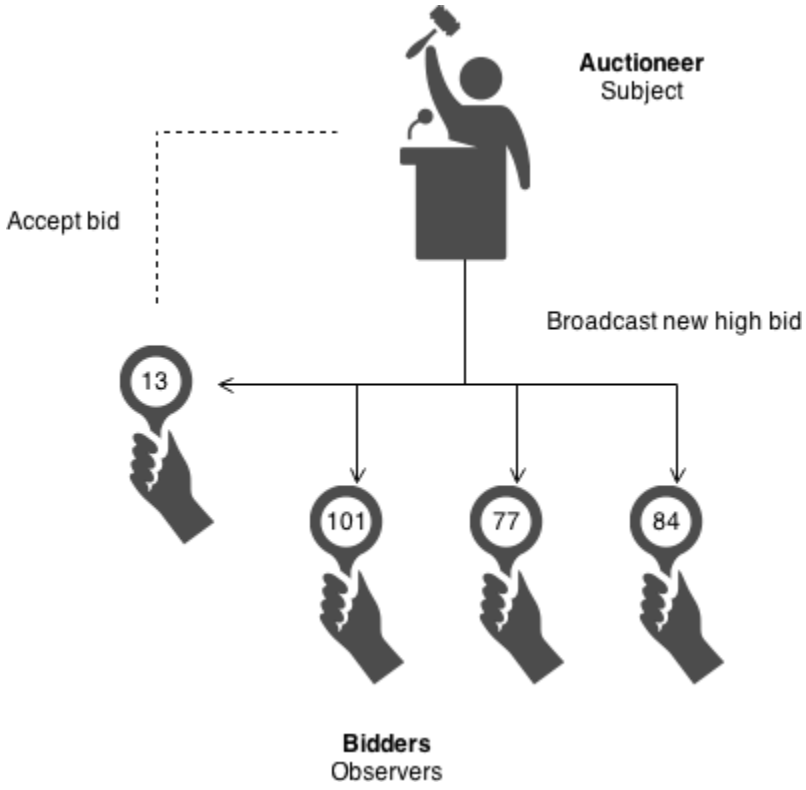
However, all bidders share a common ability: the **operation** to pay the bid if any of their bids are accepted by the auctioneer.

Auctioneer (Subject)

- minPrice : double

- bidders : List<Bidder>


+ setMinPrice(price : double)
+ attach(b : Bidder)
+ detach(b : Bidder)
+ notifyBidders()
+ acceptBid(b : Bidder, amount : double)
| 1-to-many
v
Bidder <<interface>> (Observer)
+ update(minPrice : double)
+ placeBid(amount : double)
+ payBid() : void
/|\
| (implements)
|
InPersonBidder OnlineBidder PhoneBidder
+ update(...) + update(...) + update(...)
+ placeBid(...) + placeBid(...) + placeBid(...)
+ payBid() + payBid() + payBid()



Submission instructions:

- Clone the assignment repository as a local directory.
- Written answers should be added to a single PDF file and included in the assignment repository.
- The PDF file should include your name as it appears on Canvas.
- Comment on your solution steps. Comments are essential to earn a complete score.
- Push your implementation to the online repository to submit your work.
- No points will be given if the Java files fail to compile. Remember: do not submit compiled files.