1. **Name:** Keval Shah
   GH link: https://github.com/kevalds51/keval_CSCI_5448_project

2. **Project Description** (Artifact Management Services): An inventory management and procurement application for artifacts. Organizations such as museums and galleries, manage their collection and funds by trading, lending and auctioning of their artifacts. They can also transact with individual collectors under certain conditions.

3. **Features implemented:**

ID	Requirements
UR-001	As a system administrator, I need to log in so I can monitor the system.
UR-003	As a system administrator, I want to be able to add\delete artifacts so I can keep the system updated.
UR-004	As a curator, I need to login and check artifact exchange requests.
UR-005	As a curator, I want to update the status of Museum artifacts and post Auctions.
UR-006	As a curator, I want to be able to evaluate (approve/reject) artifact sell requests from collectors.
UR-007	As a curator, I want to maintain a list of Museum transactions.
UR-008	As an auctioneer, I want to login to get auction hosting information.
UR-009	As an auctioneer, I want to see the auction and artifact sale details that has to be conducted by me.
UR-010	As an auctioneer, I want to assess the bids and declare selected buyer if any.
UR-011	As an art collector, I have to register an account and login so I can use the special features of the system.
UR-012	As an art collector, I want to search for an artifact based on artist, value, age etc.
UR-013	As an art collector, I want to know the status of that artifact and check if it is up for sale or auction.
UR-014	As an art collector, I want to know the auction details for a given artifact.
UR-015	As an art collector, I want to submit a bid of my choice and get the auction result when available.
UR-016	As an art collector, I want to propose a sale of my artifact to the Museum.
UR-017	As an art collector, I want to maintain and access my transactions with the Museum.

4. **Features not implemented:**

ID	Requirements
UR-002	As a system administrator, I need to access any user's account for some situation.

5. **Final Class diagram** *(Diagram on next page)***:** The initial class diagram was very crude and was plainly focussed on implementing the features in any possible manner. I made two significant updates till I had finished the final implementation:-

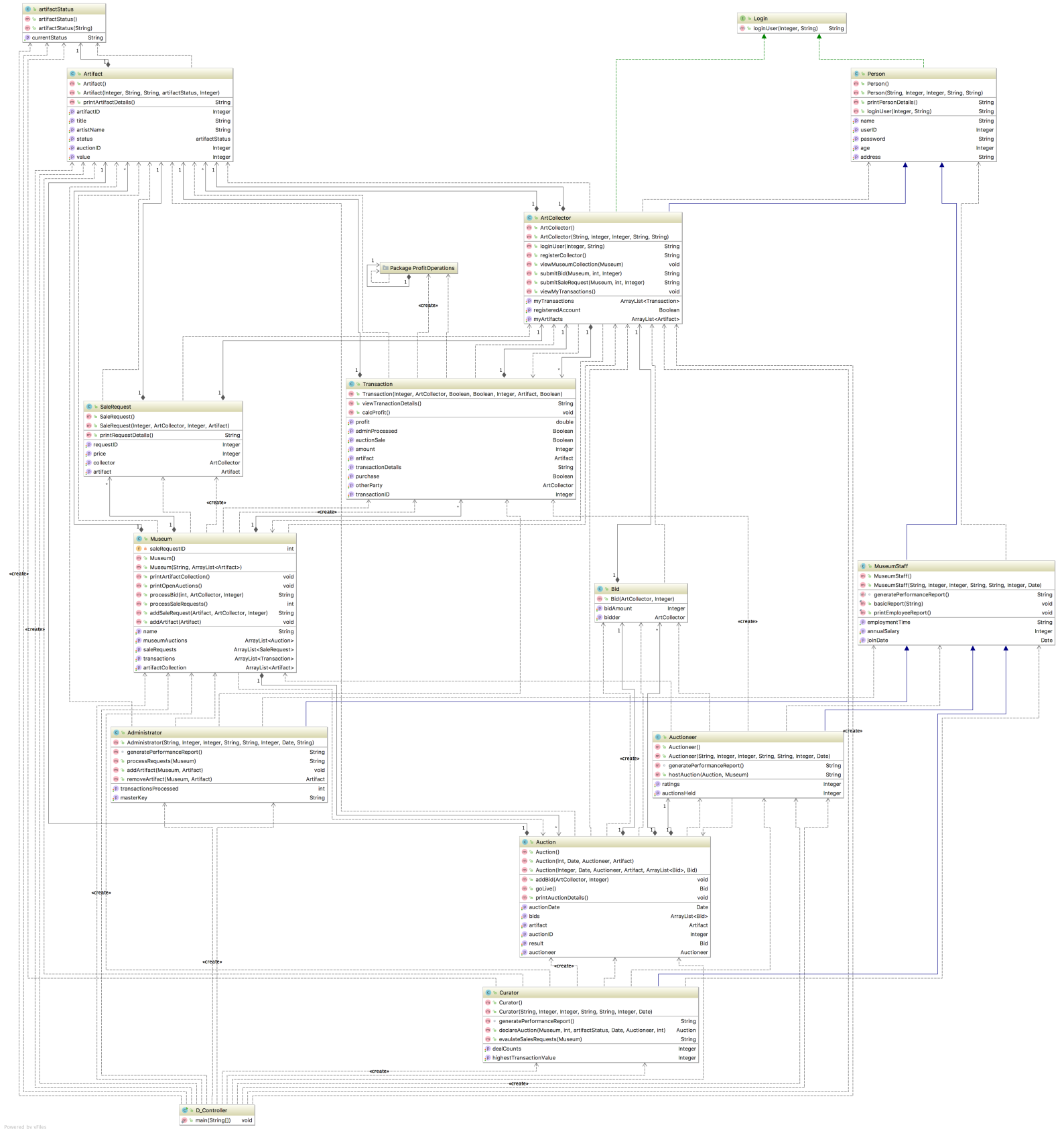
    1. The first major update was made to follow the OO conventions and practices. This involved using the right function calls, passing around the right objects as parameters and finally implementing the algorithms in the right classes. Earlier, I was not always executing the algorithm/ computation in the class to which it belonged. For example, If I was calculating the profit from selling an artifact, I was doing it in the *seller* class instance instead of the *Museum* class where it was supposed to be done in real case scenarios.

    2. The second major update was made while implementing the selected design patterns. After finding the hotspots, I had to change the structure of a few classes and reconsider the modeling. Moreover, I had to fit the parent-subclass in the format that was required for the particular design pattern. For example, while implementing the *Strategy* design pattern for the transactions class, I had to add a new method.

    Note: In the below class diagram, I have not shown the classes that were implemented as a part of the design patterns. The design pattern classes are shown in the design pattern description as mentioned the project description.

CSCI-5448: Project Part - 6

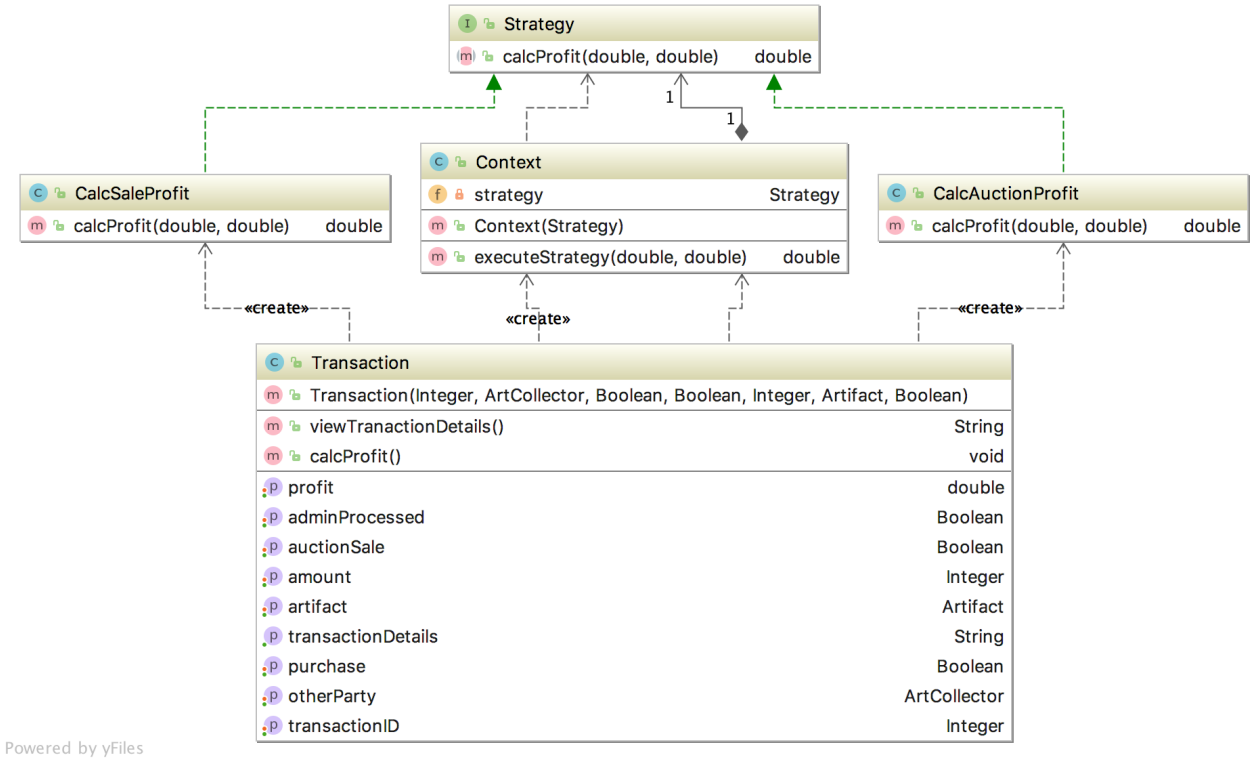
Final Class Diagram



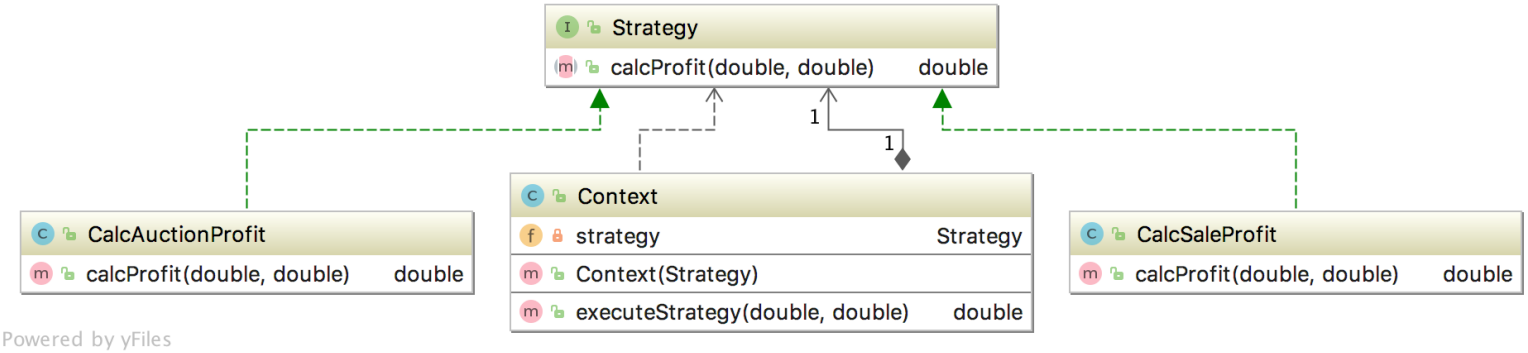
6. Design Patterns:

I. Strategy

A. *Class usage:* This design pattern has been used in the *Transactions* class to bifurcate the functioning of the method *calcProfit()*.



B. *Class diagram of the design pattern*:



C. *Why and how is it implemented:* One of the tasks of the *Transactions* class is responsible for maintaining and calculating the profit/loss of an Artifact incase of an auction sale or a direct purchase. However, the way the profit is

calculated is different based on a class property. Hence, we want to select the class behavior (i.e. the algorithm) at run time. The strategy design pattern is ideal for this implementation. In this pattern, we produce the objects that represent different strategies. We also create a context object whose behavior varies as per its strategy object. Hence, this strategy object allows us to vary the executing algorithm of the context object. For this project, The method that has been bifurcated is: *calcProfit()*. The two concrete classes implementing the strategy interface are: CalcAuctionProfit and CalcSaleProfit.

II. Template

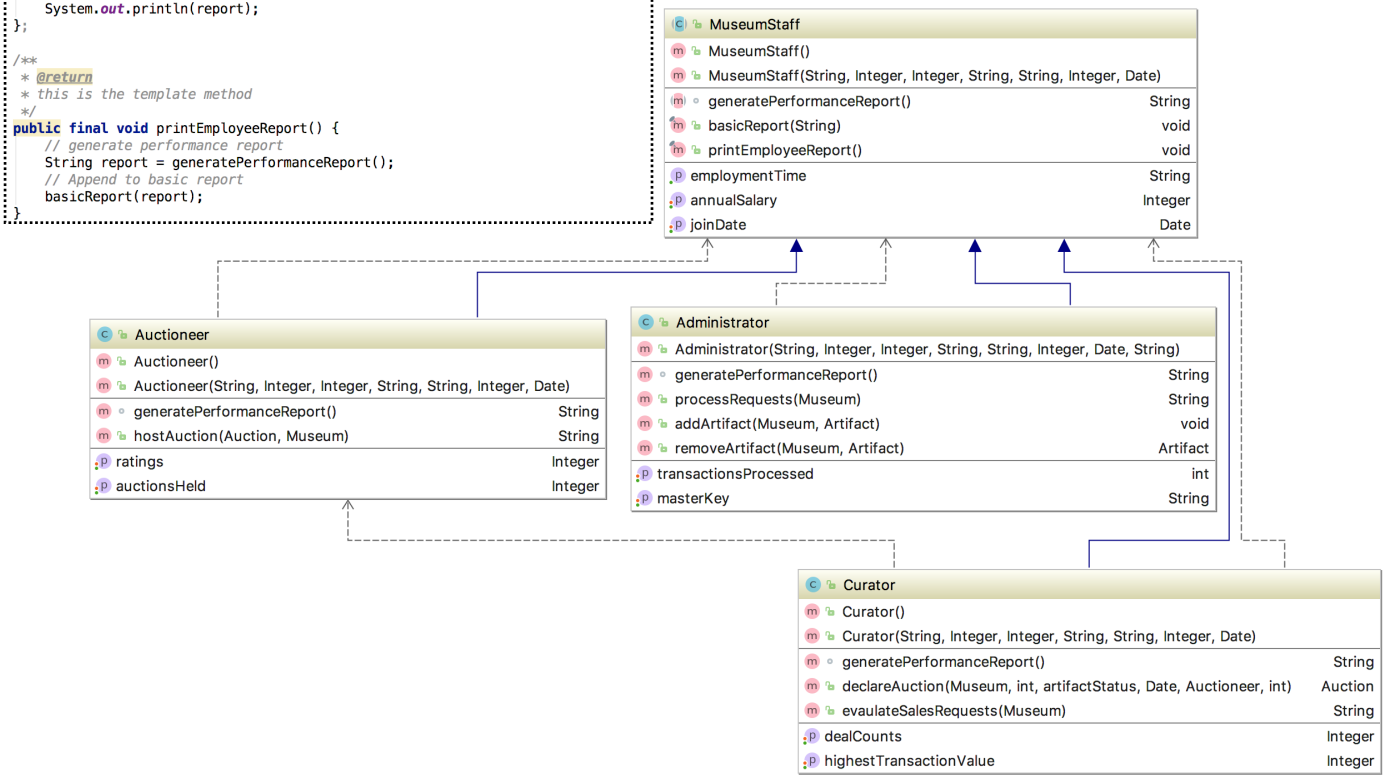
A. *Class usage:* This design pattern has been used in the *MuseumStaff* class and its subclasses: Administrator, Curator and Auctioneer (can be seen in the same class diagram for design pattern below). It is used to efficiently implement the generate *EmployeeReport()* functionality.

B. *Class diagram of the design pattern*:

```
abstract String generatePerformanceReport();

/**
 * @param report
 * @return
 */
public final void basicReport(String report){
    System.out.println("\n\t--- Employee Report for: " + this.getName()+" ---");
    System.out.println(this.getEmploymentTime());
    System.out.println(report);
};

/**
 * @return
 * this is the template method
 */
public final void printEmployeeReport() {
    // generate performance report
    String report = generatePerformanceReport();
    // Append to basic report
    basicReport(report);
}
```



C. *Why this and how is it implemented:* In the Template design pattern, a parent-abstract class exposes defined template[s] to execute its Algorithm. The concrete subclasses are required to override some of the methods based on their need. However, this invocation is required to be in the same order as in the abstract class. In this project, we want to generate the report of any type of Museum staff. There are three subclasses of the abstract *MuseumStaff*. There are employee's basic details like the employment duration, name etc that are common to all the subclasses and have to be in the report. However, each of the subclasses have different evaluation metrics. For example, an Auctioneer's report is required to have ratings and auctionsHeld, an Admin's report is required to have the transactionsProcessed and so on. Hence, we create a template called *printEmployeeReport()* for printing the final report. This will be a combination of abstract methods that vary between subclasses along with the final methods that are common to the subclasses. The abstract method that is required to be overwritten by all the concrete subclasses. Similarly, we will create a final method in the abstract *MuseumStaff* to produce the employee details in the report for any type of employee. Since the order of steps in creating the report is fixed, the template design pattern fits in efficiently.

III. **Facade**

A. *Class usage:*

B. *Class diagram of the design pattern*:

C. *Why this and how is it implemented:*