# Final Report

1. **Name: Shravan Shetty**
    a. GitHub link: https://github.com/shsh9888/shoppingcart
    b. Technologies : Spring boot, Java, MVC, Hibernate,Thymeleaf

2. **Project Description :** Implementation of general shopping cart consisting of 3 different types of users namely Customer, Seller and Admin. Each of the users have their own responsibilities. On a high-level once the system is ready any customer can login/register and buy the items which have listed by the seller. Admin acts as a super user who controls the orders.

3. **List of implemented features.**

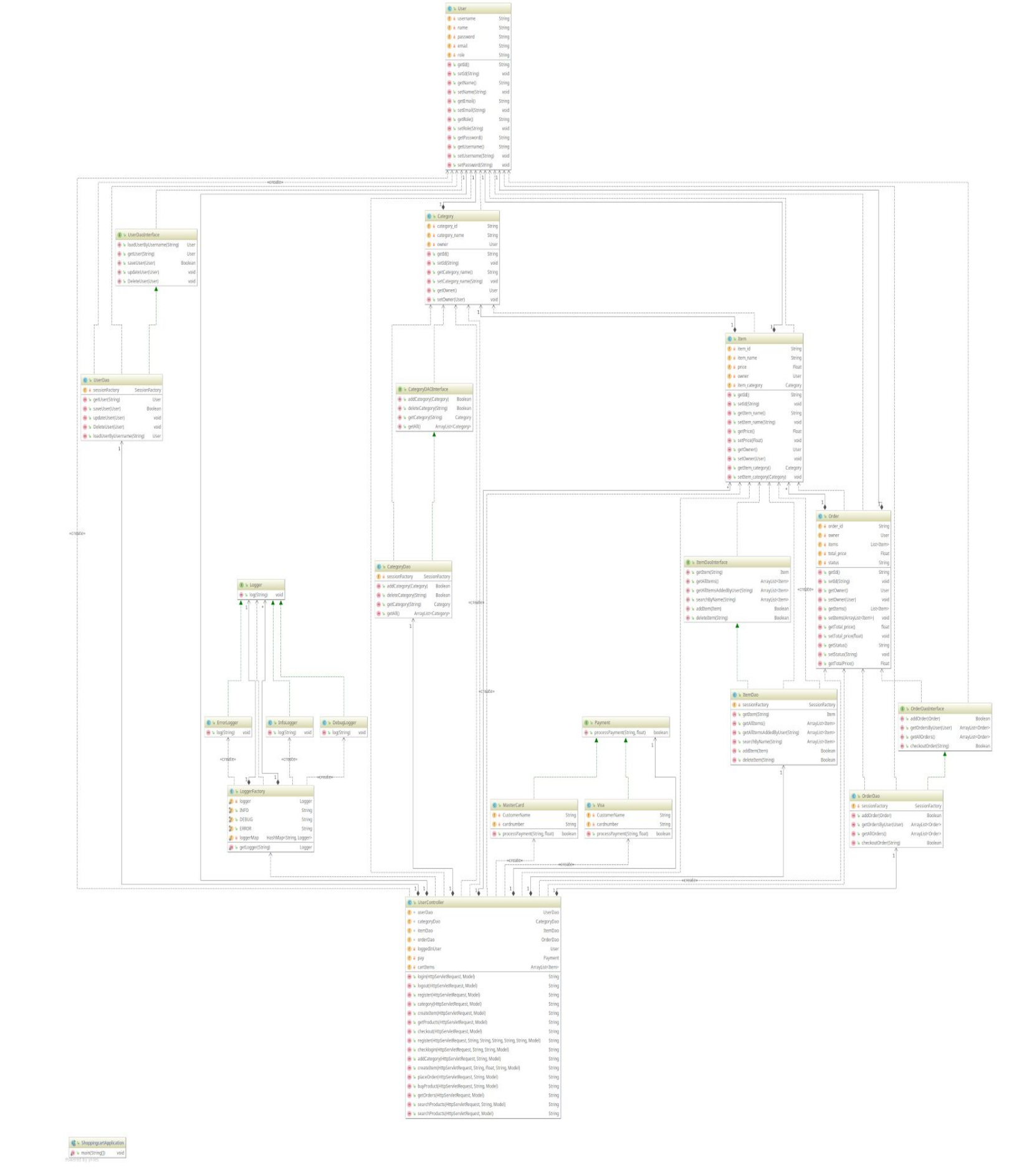
| ID | Features |
|---|---|
| 1. | Customers ,Sellers and admins can sign up/ Register |
| 2. | Customers, Sellers and admins can login |
| 3. | Sellers can categories of a product |
| 4. | Sellers can add products/items and respective prices. |
| 5. | Customers can add items to the cart |
| 6. | Customers can checkout/pay with a selected payment method. |
| 7. | Customers/Sellers search for an item by name. |
| 8. | Customers can view their previous orders |
| 9. | Admin can view all the orders |

4. Lists of features not implemented.

| ID | Features |
|---|---|
| 1. | Admin can review sellers and approve(add) or delete them. |

5. **Class Dlagram (Final):**
    a. Changes:
        i. Merged the controllers into a single UserController. The reason for this was to implement a front controller design pattern and keep Usercontroller as a front controller which handles all the Web requests. Note that this is and additions to the 3 design patterns.
        ii. Merged the different User classes (Admin, Seller, Customer) into a single user class as they did not differ much in the methods and properties.
        iii. Added a new package called payment with an Interface Payment and classes Visa and MasterCard implementing the interface. This was done to implement strategy pattern. I did not think this through during the initial design as I did not think about design patterns .
        iv. Implemented a new logging functionality internal to project with new classes like InfoLogger,DebugLogger and ErrorLogger. I did not think this through during the initial design as I did not think about design patterns.
        v. Some new methods and properties were added to the existing classes to support the implementation.

Please find the complete class diagram below. (Also towards the end links to all the images used in the report are given)

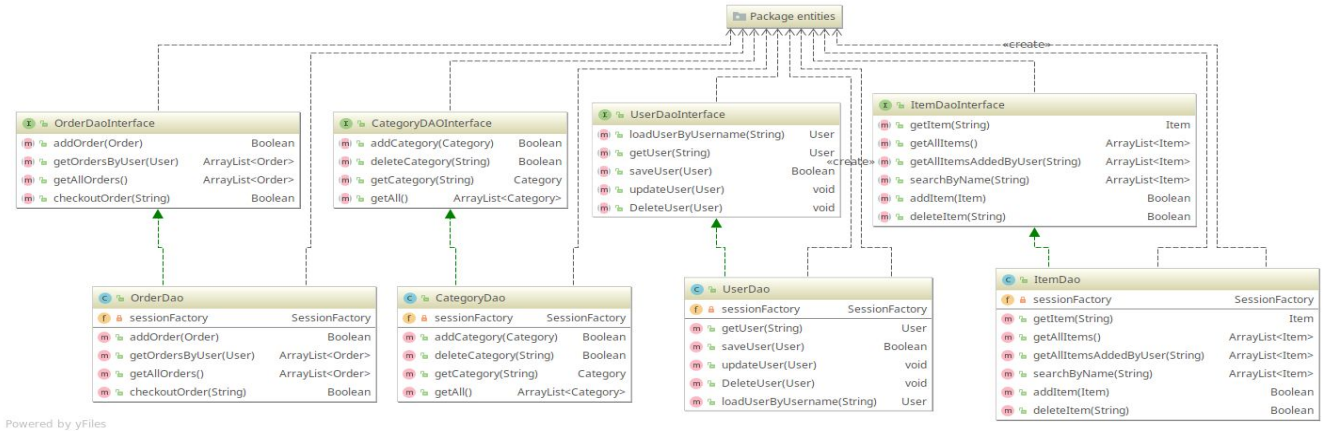

User
username String
name String
password String
email String
role String
getId() String
setId(String) void
getName() String
setName(String) void
getEmail() String
setEmail(String) void
getRole() String
setRole(String) void
getPassword() String
getUsername() String
setUsername(String) void
setPassword(String) void
«create»
UserDaoInterface
loadUserByUsername(String) User
getUser(String) User
saveUser(User) Boolean
updateUser(User) void
DeleteUser(User) void
Category
category_id String
category_name String
owner User
getId() String
setId(String) void
getCategory_name() String
setCategory_name(String) void
getOwner() User
setOwner(User) void
Item
item_id String
item_name String
price Float
owner User
item_category Category
getId() String
setId(String) void
getItem_name() String
setItem_name(String) void
getPrice() Float
setPrice(Float) void
getOwner() User
setOwner(User) void
getItem_category() Category
setItem_category(Category) void
UserDao
sessionFactory SessionFactory
getUser(String) User
saveUser(User) Boolean
updateUser(User) void
DeleteUser(User) void
loadUserByUsername(String) User
CategoryDAOInterface
addCategory(Category) Boolean
deleteCategory(String) Boolean
getCategory(String) Category
getAll() ArrayList<Category>
Order
order_id String
owner User
items List<Item>
total_price Float
status String
getId() String
setId(String) void
getOwner() User
setOwner(User) void
getItems() List<Item>
setItems(ArrayList<Item>) void
getTotal_price() float
setTotal_price(float) void
getStatus() String
setStatus(String) void
getTotalPrice() Float
«create»
CategoryDao
sessionFactory SessionFactory
addCategory(Category) Boolean
deleteCategory(String) Boolean
getCategory(String) Category
getAll() ArrayList<Category>
ItemDaoInterface
getItem(String) Item
getAllItems() ArrayList<Item>
getAllItemsAddedByUser(String) ArrayList<Item>
searchByName(String) ArrayList<Item>
addItem(Item) Boolean
deleteItem(String) Boolean
Logger
log(String) void
ItemDao
sessionFactory SessionFactory
getItem(String) Item
getAllItems() ArrayList<Item>
getAllItemsAddedByUser(String) ArrayList<Item>
searchByName(String) ArrayList<Item>
addItem(Item) Boolean
deleteItem(String) Boolean
OrderDaoInterface
addOrder(Order) Boolean
getOrdersByUser(User) ArrayList<Order>
getAllOrders() ArrayList<Order>
checkoutOrder(String) Boolean
ErrorLogger
log(String) void
InfoLogger
log(String) void
DebugLogger
log(String) void
Payment
processPayment(String, float) boolean
«create»
LoggerFactory
logger Logger
INFO String
DEBUG String
ERROR String
loggerMap HashMap<String, Logger>
getLogger(String) Logger
MasterCard
CustomerName String
cardnumber String
processPayment(String, float) boolean
Visa
CustomerName String
cardnumber String
processPayment(String, float) boolean
OrderDao
sessionFactory SessionFactory
addOrder(Order) Boolean
getOrdersByUser(User) ArrayList<Order>
getAllOrders() ArrayList<Order>
checkoutOrder(String) Boolean
UserController
userDao UserDao
categoryDao CategoryDao
itemDao ItemDao
orderDao OrderDao
loggedInUser User
pay Payment
cartItems ArrayList<Item>
login(HttpServletRequest, Model) String
logout(HttpServletRequest, Model) String
register(HttpServletRequest, Model) String
category(HttpServletRequest, Model) String
createItem(HttpServletRequest, Model) String
getProducts(HttpServletRequest, Model) String
checkout(HttpServletRequest, Model) String
register(HttpServletRequest, String, String, String, String, String, Model) String
checklogin(HttpServletRequest, String, String, Model) String
addCategory(HttpServletRequest, String, Model) String
createItem(HttpServletRequest, String, Float, String, Model) String
placeOrder(HttpServletRequest, String, Model) String
buyProduct(HttpServletRequest, String, Model) String
getOrders(HttpServletRequest, Model) String
searchProducts(HttpServletRequest, String, Model) String
searchProducts(HttpServletRequest, Model) String
ShoppingcartApplication
main(String[]) void

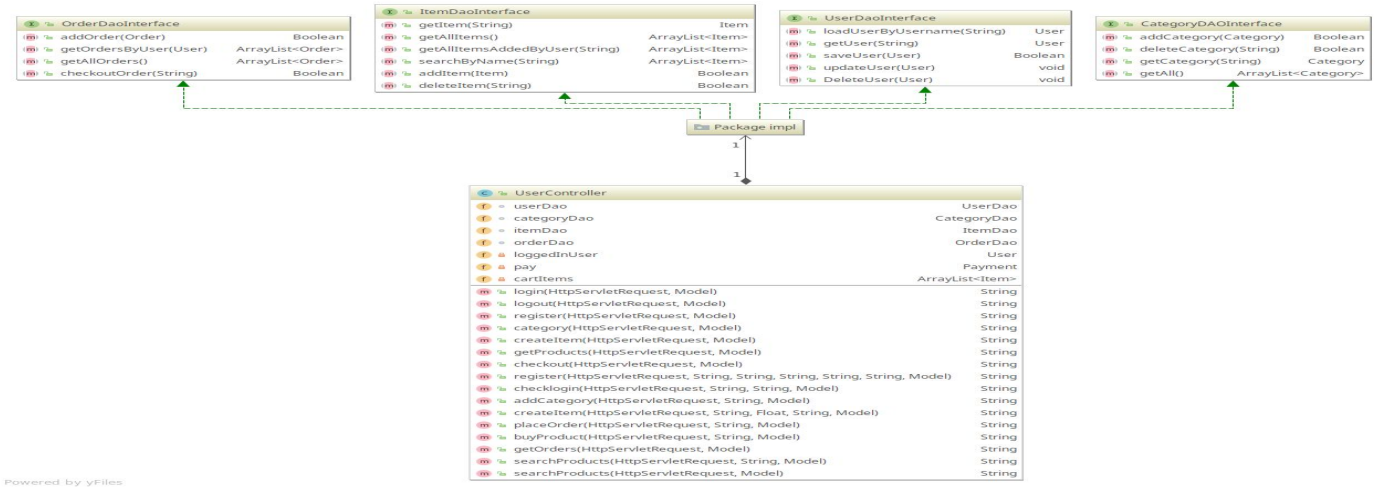
## 6. Design Patterns

a. **Data Access Object Pattern (DAO):** The DAO pattern allows to isolate the application layer from the persistence layer. I have used ORM in my project with the help of Hibernate which exposes database as entity classes. This allowed me to separate all the DB operations needed by the project. I used this design pattern because of the one of the core principles of the pattern itself (separation of application layer from persistence layer). In addition to that if I am extending the project and change the database from MYSQL to MONGO only this layer would require the change and application layer remains as it. This aligns with open for extension and closed for modification principle.

I implemented DAO pattern by creating interfaces for each entity object and listing down all the operations with regards to that. Post that there is a Dao class corresponding to this which will implement all these methods. This class object will be used in the UserController to do db operations based on the request.

Please find the design pattern diagram followed by Usage diagram below.

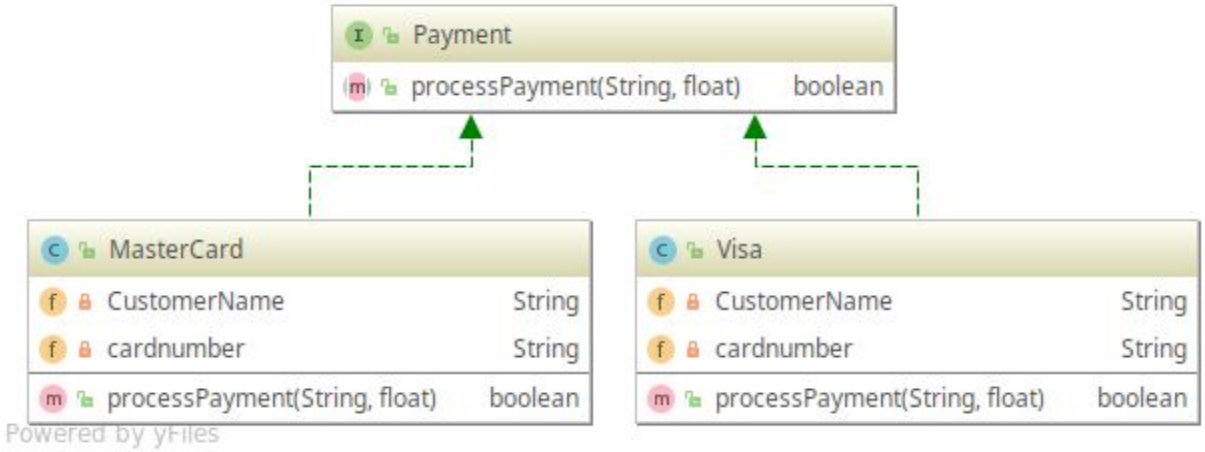


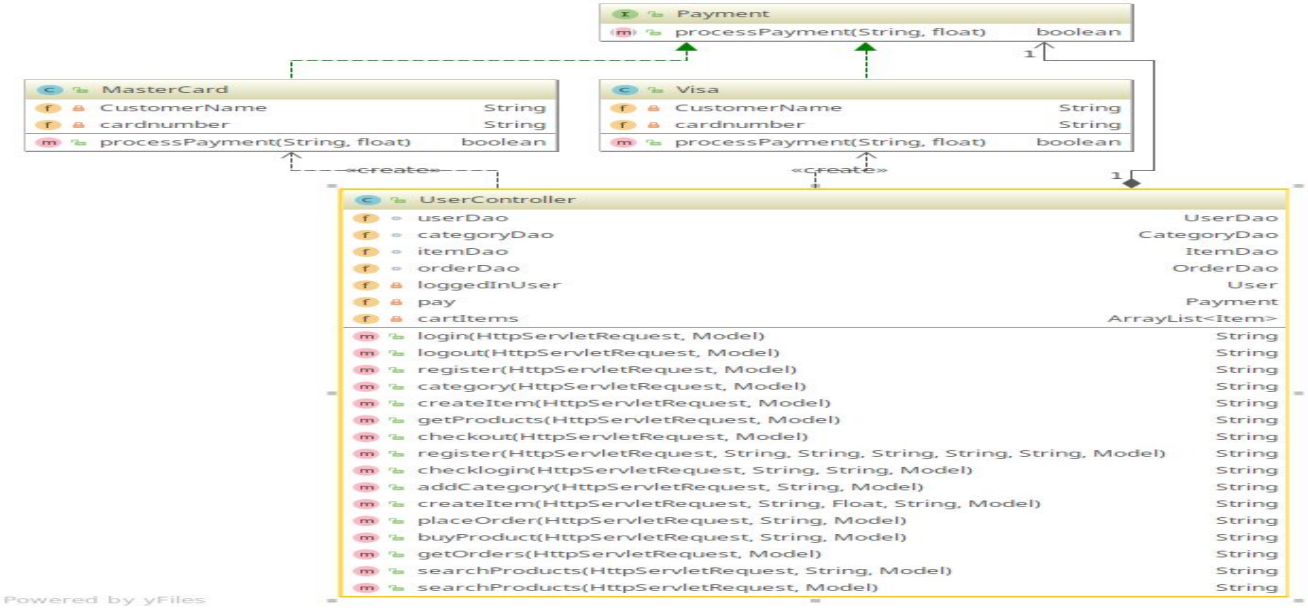



b. **Strategy Pattern** : Strategy is one of the design patterns used when we have to choose the algorithm at the run time. I came across a payment use case in my project where user can choose multiple ways to pay for the order. Usually every customer chooses the payment option towards the end during checkout which means until then program is not aware what algorithm or module to call to process the request. I found this use case ideal for Str

For simplicity I have just used Visa and MasterCard for this implementation with a Payment as main interface and Visa and Mastercard will implement the processPayment() Method. When the payment request arrives in the UserController it decides at the run time which one to use using polymorphism.

Please find the design pattern diagram followed by Usage diagram below.

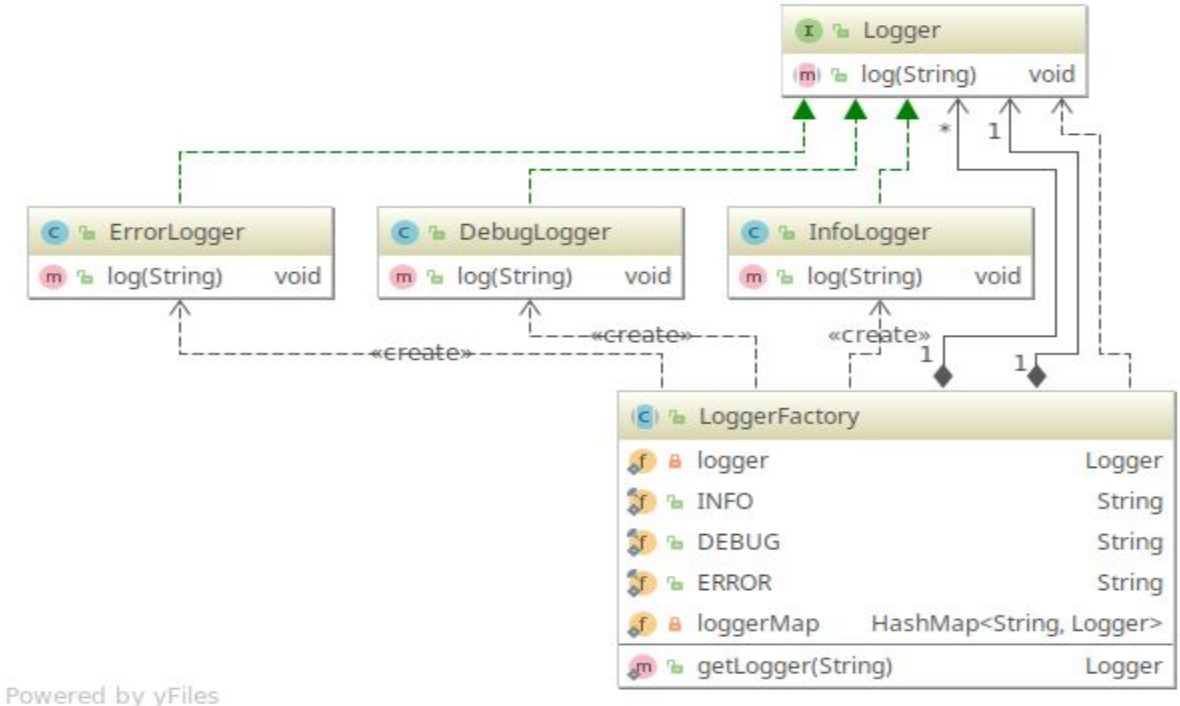


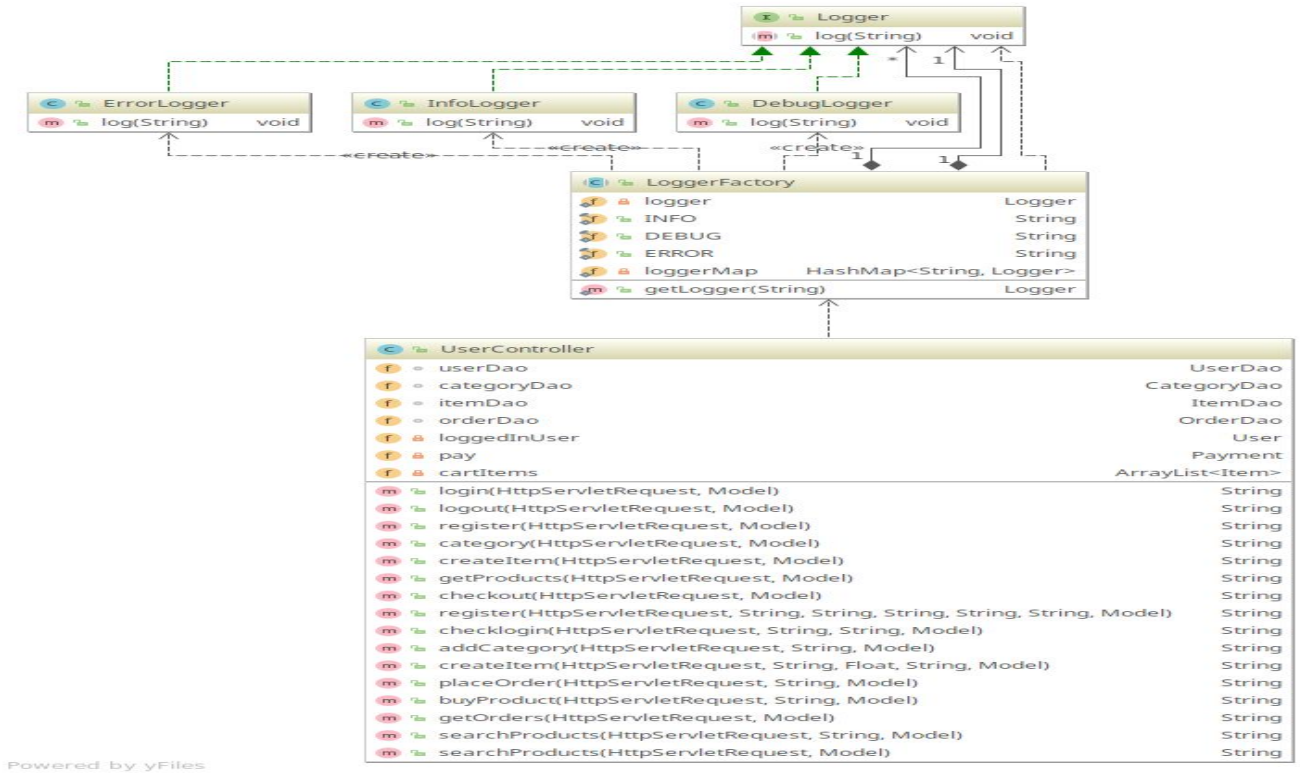



c. **Factory Pattern :** In the Factory pattern we create objects without knowing the creation logic of the objects. I used Factory to implement the logging functionality. As the project got slightly bigger it was hard to debug any issues related to the features in the project. So I needed different types of loggers namely Info, Debug and Error which I can use to write to console to see what exactly is happening inside the system. These loggers can have different types of messages and colors to help and understand debug. They can even write to different locations.

I created a Logger interface which exposed the method to log. Infologger, DebugLogger and ErrorLogger implemented this logger. As there is no point in creating new logger objects each time, I created a loggerfactory with a map which keeps track of the created objects and returns the same if its already been created.

Please find the design pattern diagram followed by Usage diagram below.






## 7. Learnings from the project

- **a.** Incremental design and implementation of the project. This really helped in retrospecting all the previous steps before going ahead implementing the current one.
- **b.** Project kept getting finer as I went through each step and corrected the mistakes.
- **c.** Having all the functionalities and requirements jolted down initially helped me to decide on the techstack early and effectively.
- **d.** Deciding on the classes in the start and designing the class diagram helped to figure out the relationship between classes. This helped a lot when I started coding as I already knew what I had to implement.
- **e.** Also having the class diagrams initially helped me to design the DB and there were no changes to that after that.
- **f.** Learned how to use MVC pattern effectively which separated the responsibilities and increased readability of the code and made it easier to debug.
- **g.** Using Hibernate helped me learn and implement DAO pattern which in itself can be extended.
- **h.** Implementing custom logger with factory design pattern helped to debug issues.

8. **Miscellaneous:**
    a. **Clear Image Links:**
        i. Final Class Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/Final_class_diagram_shoppingcart.png
        ii. DAO Pattern Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/DAO_Pattern.png
        iii. DAO Pattern Usage
        https://github.com/shsh9888/shoppingcart/blob/master/images/Dao_Pattern_Usage.png
        iv. Strategy Pattern Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/Strategy_Pattern.png
        v. Strategy Pattern Usage Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/Strategy_Pattern_Usage.png
        vi. Factory Pattern Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/Factory_Pattern.png
        vii. Factory Pattern Usage Diagram
        https://github.com/shsh9888/shoppingcart/blob/master/images/FactoryPattern_Usage.png

    b. **Auto Generated Code**
    https://github.com/shsh9888/shoppingcart/tree/master/Shetty_Shoppingcart_Part6Code
    c. **Presentation**
    https://github.com/shsh9888/shoppingcart/tree/master/Presentation