Team: Kristopher Neilsen, Samuel Steward, Chandler Baggett

Title: Restaurant Ordering System

Project Part 6: Final Report

1. List the features that were implemented (table with ID and title).

Features Implemented - User Requirements				
ID	**Requirements**	**Topic Area**	**Actor**	**Priority**
UR-001	As a Customer, I want to be able to create an account so that I can personalize the ordering experience.	Account Management	Customer	High
UR-002	As an Admin I want to be able to add/remove Staff Accounts so that I can have control over staff that use the system.	Account Management	Admin	High
UR-003	Users should be able to login to and logout of the system	Account Management	Customer, Staff, Admin	High
UR-004	As a Customer I want to be able to view the menu of the restaurant so that I can create an order.	Menu Viewing	Customer	High
UR-005	As a Customer I want to be able to view the menu item prices on the menu so that I know the price of each item.	Menu Viewing	Customer	High
UR-006	As an Admin I want to be able to add/remove menu items	Menu Management	Admin	High
UR-008	As a Customer I want to be able to create a new order so that I can group together items that I wish to order from the Restaurant.	Order Management	Customer	High
UR-009	As a Customer I want to be	Order	Customer	High

	able to add/remove menu items to/from my order.	Management		
UR-010	As a Customer I want to be able to submit my order to the restaurant so that they can view it.	Order Management	Customer	High
UR-011	As a Customer I want to be able to view the order total so that I know how much I'm spending when I submit the order	Order Management	Customer	High
UR-012	As a Customer I want to be able to save an in-progress order so that I can resume the order later.	Order Management	Customer	Medium
UR-013	As a Customer I want to be able to resume an in-progress order so that I can come back later to complete my order.	Order Management	Customer	Medium
UR-014	As a Customer I want to be able to cancel an in-progress order so that I can start a new order.	Order Management	Customer	Medium
UR-015	As a Staff or Admin I want to be able to mark orders submitted to the restaurant as complete	Order Management	Staff, Admin	High
UR-016	As a Staff or Admin I want to be able to view submitted but not completed orders	Order Viewing	Staff, Admin	High
UR-017	As a Staff or Admin I want to be able to see the full order history for the restaurant.	Order Viewing	Staff, Admin	High
UR-018	As a Customer, I want to be able to view my full order history, so that I can see what I ate in the past.	Order Viewing	Customer	Medium

2. List the features were not implemented from Part 2 (table with ID and title).

Features Not Implemented - User Requirements				
ID	**Requirements**	**Topic Area**	**Actor**	**Priority**
UR-007	As an Admin I want to be able to edit the menu items.	Menu Management	Admin	High

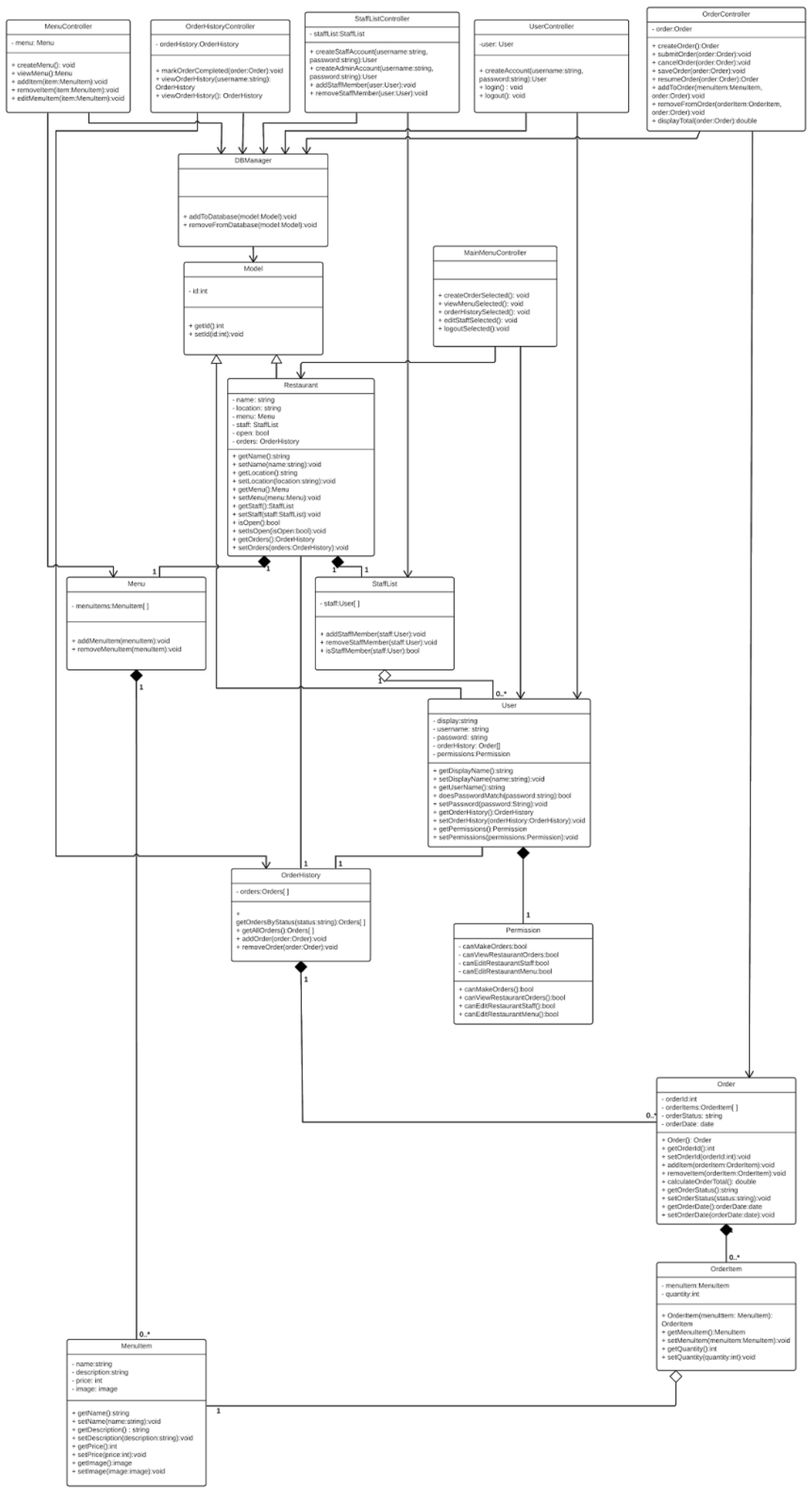
3. Show your Part 2 class diagram and your final class diagram.
What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

Due to the export quality of LucidCharts, some of our images appear blurry. The links below can be used to view a clearer image of our diagrams.

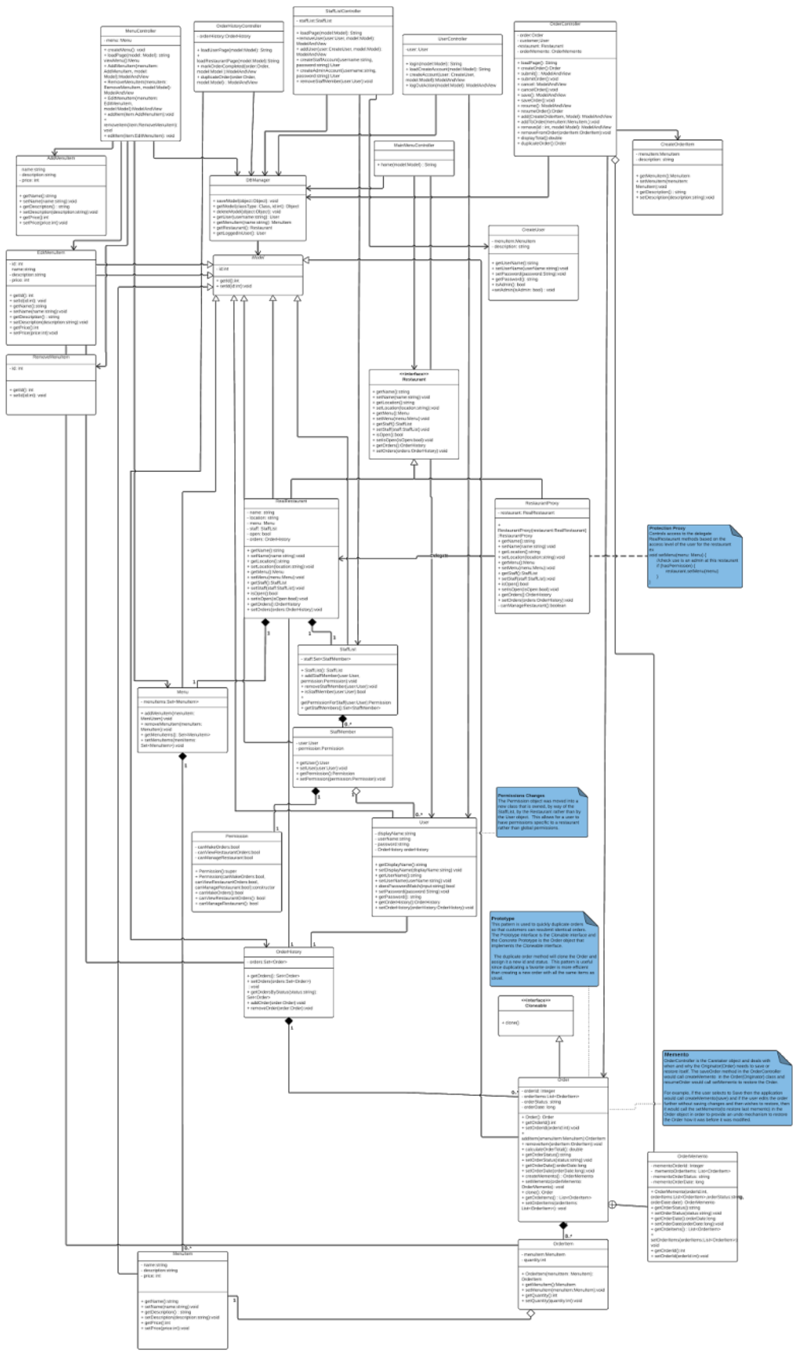
Part 2 Class Diagram:
https://www.lucidchart.com/publicSegments/view/dc15acb6-8710-4a55-8f7f-6b28ca83959f/image.png

Final Class Diagram:
https://www.lucidchart.com/documents/view/dc0db50a-d434-420f-9df7-d2e88a86e3a0

Part 2 Class Diagram:


MenuController
- menu: Menu
+ createMenu(): void
+ viewMenu():Menu
+ addItem(item:MenuItem):void
+ removeItem(item:MenuItem):void
+ editMenuItem(item:MenuItem):void
OrderHistoryController
- orderHistory:OrderHistory
+ markOrderCompleted(order:Order):void
+ viewOrderHistory(username:string): OrderHistory
+ viewOrderHistory(): OrderHistory
StaffListController
- staffList:StaffList
+ createStaffAccount(username:string, password:string):User
+ createAdminAccount(username:string, password:string):User
+ addStaffMember(user:User):void
+ removeStaffMember(user:User):void
UserController
-user: User
+ createAccount(username:string, password:string):User
+ login() : void
+ logout(): void
OrderController
- order:Order
+ createOrder():Order
+ submitOrder(order:Order):void
+ cancelOrder(order:Order):void
+ saveOrder(order:Order):void
+ resumeOrder(order:Order):Order
+ addToOrder(menuItem:MenuItem, order:Order):void
+ removeFromOrder(orderItem:OrderItem, order:Order):void
+ displayTotal(order:Order):double
DBManager
+ addToDatabase(model:Model):void
+ removeFromDatabase(model:Model):void
MainMenuController
+ createOrderSelected(): void
+ viewMenuSelected(): void
+ orderHistorySelected(): void
+ editStaffSelected(): void
+ logoutSelected():void
Model
- id:int
+ getId():int
+ setId(id:int):void
Restaurant
- name: string
- location: string
- menu: Menu
- staff: StaffList
- open: bool
- orders: OrderHistory
+ getName():string
+ setName(name:string):void
+ getLocation():string
+ setLocation(location:string):void
+ getMenu():Menu
+ setMenu(menu:Menu):void
+ getStaff():StaffList
+ setStaff(staff:StaffList):void
+ isOpen():bool
+ setIsOpen(isOpen:bool):void
+ getOrders():OrderHistory
+ setOrders(orders:OrderHistory):void
1
1
1
1
Menu
- menuItems:MenuItem[ ]
+ addMenuItem(menuItem):void
+ removeMenuItem(menuItem):void
1
StaffList
- staff:User[ ]
+ addStaffMember(staff:User):void
+ removeStaffMember(staff:User):void
+ isStaffMember(staff:User):bool
1
0..*
User
- display:string
- username: string
- password: string
- orderHistory: Order[]
- permissions:Permission
+ getDisplayName():string
+ setDisplayName(name:string):void
+ getUserName():string
+ doesPasswordMatch(password:string):bool
+ setPassword(password:String):void
+ getOrderHistory():OrderHistory
+ setOrderHistory(orderHistory:OrderHistory):void
+ getPermissions():Permission
+ setPermissions(permissions:Permission):void
1
1
OrderHistory
- orders:Orders[ ]
+ getOrdersByStatus(status:string):Orders[ ]
+ getAllOrders():Orders[ ]
+ addOrder(order:Order):void
+ removeOrder(order:Order):void
1
1
Permission
- canMakeOrders:bool
- canViewRestaurantOrders:bool
- canEditRestaurantStaff:bool
- canEditRestaurantMenu:bool
+ canMakeOrders():bool
+ canViewRestaurantOrders():bool
+ canEditRestaurantStaff():bool
+ canEditRestaurantMenu():bool
0..*
Order
- orderId:int
- orderItems:OrderItem[ ]
- orderStatus: string
- orderDate: date
+ Order(): Order
+ getOrderId():int
+ setOrderId(orderId:int):void
+ addItem(orderItem:OrderItem):void
+ removeItem(orderItem:OrderItem):void
+ calculateOrderTotal(): double
+ getOrderStatus():string
+ setOrderStatus(status:string):void
+ getOrderDate():orderDate:date
+ setOrderDate(orderDate:date):void
0..*
OrderItem
- menuItem:MenuItem
- quantity:int
+ OrderItem(menuItem: MenuItem): OrderItem
+ getMenuItem():MenuItem
+ setMenuItem(menuItem:MenuItem):void
+ getQuantity():int
+ setQuantity(quantity:int):void
0..*
MenuItem
- name:string
- description:string
- price: int
- image: image
+ getName():string
+ setName(name:string):void
+ getDescription() : string
+ setDescription(description:string):void
+ getPrice():int
+ setPrice(price:int):void
+ getImage():image
+ setImage(image:image):void
1

Final Class Diagram:



Protection Proxy
Controls access to the delegate RealRestaurant methods based on the access level of the user for the restaurant
ex.
void setMenu(menu: Menu) {
    //check use is an admin at this restaurant
    if (hasPermission) {
        restaurant.setMenu(menu)
    }
}

Permissions Changes
The Permission object was moved into a new class that is owned, by way of the StaffList, by the Restaurant rather than by the User object. This allows for a user to have permissions specific to a restaurant rather than global permissions.

Prototype
This pattern is used to quickly duplicate orders so that customers can resubmit identical orders. The Prototype interface is the Clonable interface and the Concrete Prototype is the Order object that implements the Cloneable interface.

The duplicate order method will clone the Order and assign it a new id and status. This pattern is useful since duplicating a favorite order is more efficient than creating a new order with all the same items as usual.

Memento
OrderController is the Caretaker object and deals with when and why the Originator(Order) needs to save or restore itself. The saveOrder method in the OrderController would call createMemento in the Order(Originator) class and resumeOrder would call setMemento to restore the Order.

For example, if the user selects to Save then the application would call createMemento(save) and if the user edits the order further without saving changes and then wishes to restore, then it would call the setMemento(to restore last memento) in the Order object in order to provide an undo mechanism to restore the Order how it was before it was modified.

While the core of the controllers and models from the original class diagram are still present, the final version has a number of new additions and modifications that significantly increase its complexity. First, several design patterns were added, specifically the memento, proxy and prototype patterns, which are expanded upon below. Second, all of the data storage models in the original design were made subclasses of the Model class, which allowed for easier addition to the database via Hibernate. Third, a number of data transfer classes were added for communication between the view (jsp files) and the java controllers. Finally a number of classes had private methods added to them in order to organize the internal code.

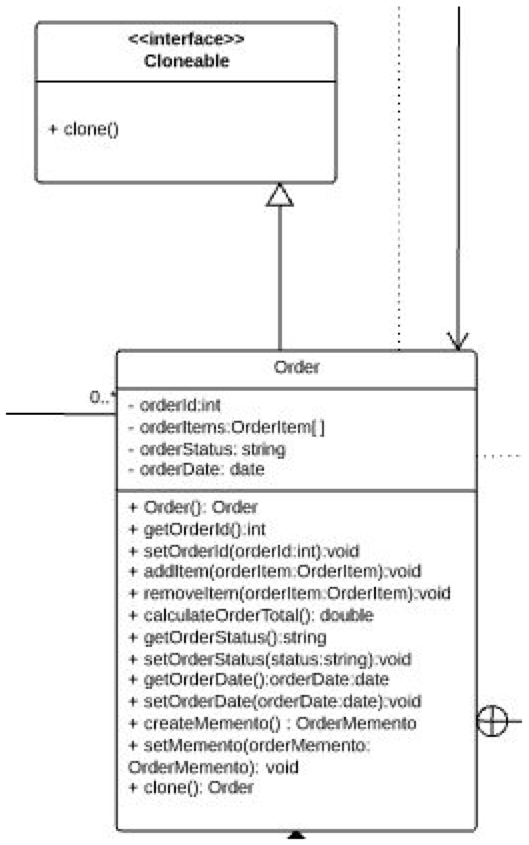
How did the original design help with the system?
The original design really gave us a solid Model View Controller architecture that we leveraged in implementing of our classes. For the most part we stuck to the original design and did not have to make major modifications once we started implementing the design.

4. Did you make use of any design patterns in the implementation of your final Prototype? If so, how? Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF).
If not, where could you make use of design patterns in your system? Show a class diagram of how you could implement each design pattern and compare how it would change from your current class diagram.

We were able to implement the following design patterns in our final project:

Prototype:

We implemented the Prototype design pattern which allows the copy of an object in order to create a new object, essentially a clone. The prototype design pattern is useful for various reasons, but the reason we used it is because we have a use case of duplicating an order(duplicate object that is required is similar to existing object).
This pattern is used to quickly duplicate orders so that customers can re-submit identical orders. The Prototype interface is the Cloneable interface and the Concrete Prototype is the Order object that implements the Cloneable interface. The duplicate order method will clone the Order and assign it a new id and status. This pattern is useful since duplicating a favorite order is more efficient than creating a new order with all the same items as usual.

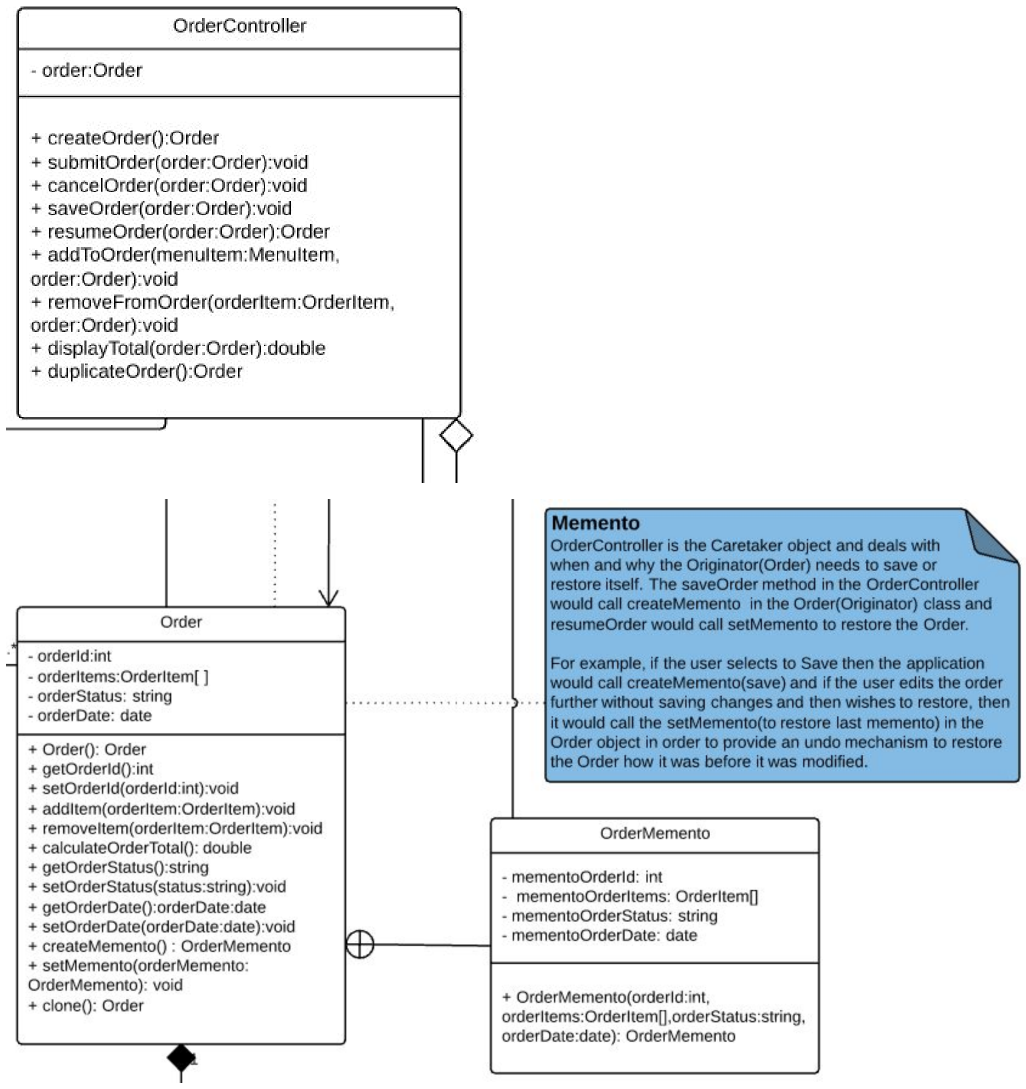

<<interface>>
Cloneable
+ clone()
Order
0..*
- orderId:int
- orderItems:OrderItem[ ]
- orderStatus: string
- orderDate: date
+ Order(): Order
+ getOrderId():int
+ setOrderId(orderId:int):void
+ addItem(orderItem:OrderItem):void
+ removeItem(orderItem:OrderItem):void
+ calculateOrderTotal(): double
+ getOrderStatus():string
+ setOrderStatus(status:string):void
+ getOrderDate():orderDate:date
+ setOrderDate(orderDate:date):void
+ createMemento() : OrderMemento
+ setMemento(orderMemento: OrderMemento): void
+ clone(): Order

Memento:

We implemented the Memento design pattern which allows an object state to be saved so that it can be reverted back to later, essentially allowing a restore or savepoint.

The Memento is applied as follows in our project; the OrderController class is the Caretaker object and deals with when and why the Originator(the Order class) needs to save or restore itself. The saveOrder method in the OrderController calls the createMemento method in the Order(Originator) class to create a Memento, or savepoint with the Order instance information.

Then if an Order is edited further and can be reverted back to the savepoint. The resumeOrder method in OrderController calls the setMemento method in the Order class to restore the Order back to that savepoint. For example, if the user selects to Save then the application calls createMemento() and if the user modifies the order without saving changes and wishes to revert back, the Restore button will call setMemento(to restore last memento) in the Order object in order to provide an undo mechanism to restore the Order how it was before it was further modified.


OrderController
- order:Order
+ createOrder():Order
+ submitOrder(order:Order):void
+ cancelOrder(order:Order):void
+ saveOrder(order:Order):void
+ resumeOrder(order:Order):Order
+ addToOrder(menuItem:MenuItem, order:Order):void
+ removeFromOrder(orderItem:OrderItem, order:Order):void
+ displayTotal(order:Order):double
+ duplicateOrder():Order
Order
- orderId:int
- orderItems:OrderItem[ ]
- orderStatus: string
- orderDate: date
+ Order(): Order
+ getOrderId():int
+ setOrderId(orderId:int):void
+ addItem(orderItem:OrderItem):void
+ removeItem(orderItem:OrderItem):void
+ calculateOrderTotal(): double
+ getOrderStatus():string
+ setOrderStatus(status:string):void
+ getOrderDate():orderDate:date
+ setOrderDate(orderDate:date):void
+ createMemento() : OrderMemento
+ setMemento(orderMemento: OrderMemento): void
+ clone(): Order
OrderMemento
- mementoOrderId: int
- mementoOrderItems: OrderItem[]
- mementoOrderStatus: string
- mementoOrderDate: date
+ OrderMemento(orderId:int, orderItems:OrderItem[],orderStatus:string, orderDate:date): OrderMemento

Memento
OrderController is the Caretaker object and deals with when and why the Originator(Order) needs to save or restore itself. The saveOrder method in the OrderController would call createMemento in the Order(Originator) class and resumeOrder would call setMemento to restore the Order.

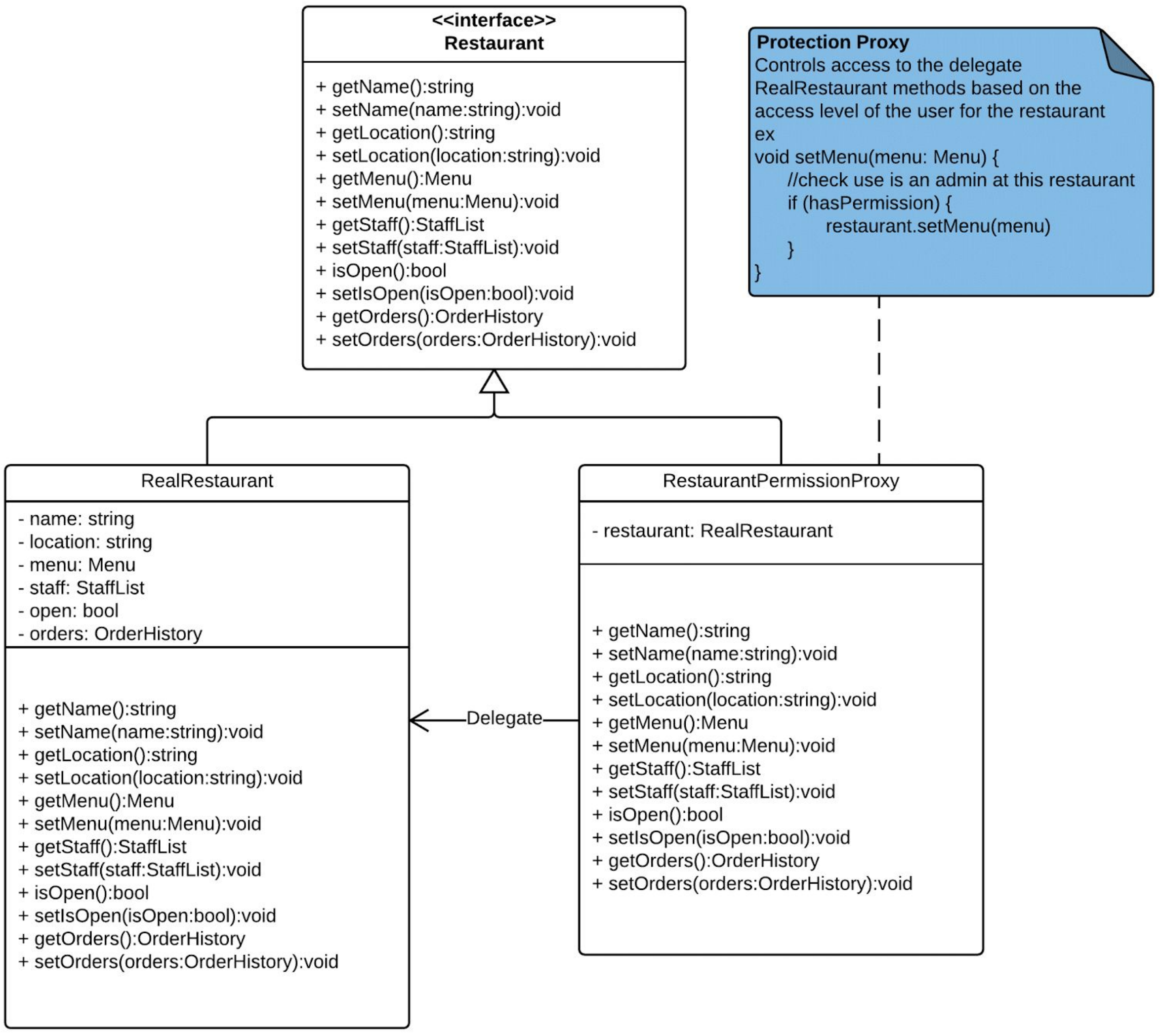
For example, if the user selects to Save then the application would call createMemento(save) and if the user edits the order further without saving changes and then wishes to restore, then it would call the setMemento(to restore last memento) in the Order object in order to provide an undo mechanism to restore the Order how it was before it was modified.

Protection Proxy:

We implemented the (Protection) Proxy design pattern, which allows for the controlling of access to methods and fields of a wrapped class, in this case used to ensure that those without permission are unable to modify data associated with a restaurant.

In our project the Proxy pattern is implemented using the Restaurant interface as the subject, the RestaurantProxy as the proxy, and the RealRestaurant as the real subject. All interactions with this proxied system take place against the Restaurant interface, no class besides the Proxy and the database know of the existence of the RealRestaurant class. The general accessor by which the entire project access the restaurant from the DB returns a RestaurantProxy, ensuring that no class can directly access a RealRestaurant without going through the RestaurantProxy.

The restaurant proxy itself uses its access to the staff of the restaurant and the knowledge of which user is logged in to ensure that no user who is not a staff member with appropriate permissions can access specified fields on the RealRestaurant, and cannot use the setters for those fields. For example, only staff for the restaurant can access the list of orders the restaurant has received, so if a customer attempts to get this information, the RestaurantProxy will simply return null instead of giving the order history.



<<interface>>
Restaurant
+ getName():string
+ setName(name:string):void
+ getLocation():string
+ setLocation(location:string):void
+ getMenu():Menu
+ setMenu(menu:Menu):void
+ getStaff():StaffList
+ setStaff(staff:StaffList):void
+ isOpen():bool
+ setIsOpen(isOpen:bool):void
+ getOrders():OrderHistory
+ setOrders(orders:OrderHistory):void
Protection Proxy
Controls access to the delegate RealRestaurant methods based on the access level of the user for the restaurant
ex
RealRestaurant
- name: string
- location: string
- menu: Menu
- staff: StaffList
- open: bool
- orders: OrderHistory
+ getName():string
+ setName(name:string):void
+ getLocation():string
+ setLocation(location:string):void
+ getMenu():Menu
+ setMenu(menu:Menu):void
+ getStaff():StaffList
+ setStaff(staff:StaffList):void
+ isOpen():bool
+ setIsOpen(isOpen:bool):void
+ getOrders():OrderHistory
+ setOrders(orders:OrderHistory):void
Delegate
RestaurantPermissionProxy
- restaurant: RealRestaurant
+ getName():string
+ setName(name:string):void
+ getLocation():string
+ setLocation(location:string):void
+ getMenu():Menu
+ setMenu(menu:Menu):void
+ getStaff():StaffList
+ setStaff(staff:StaffList):void
+ isOpen():bool
+ setIsOpen(isOpen:bool):void
+ getOrders():OrderHistory
+ setOrders(orders:OrderHistory):void

```
void setMenu(menu: Menu) {
    //check use is an admin at this restaurant
    if (hasPermission) {
        restaurant.setMenu(menu)
    }
}
```

5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

It is extremely important to spend the time up front to analyze the requirements of the system by creating different documents and diagrams; not only so the requirements can be fully analyzed, but because these resources are essential to the design of the system. During development, these documents can and should be leaned upon first to resolve any roadblocks. With that being said, there also might be changes to the design due to evolving requirements or flaws in the initial requirements analysis or design. Additionally, a good source control tool is essential, especially when multiple contributors and branches are required for working on the same code base. This makes it easier for handling branching, code merges, and code reviews.