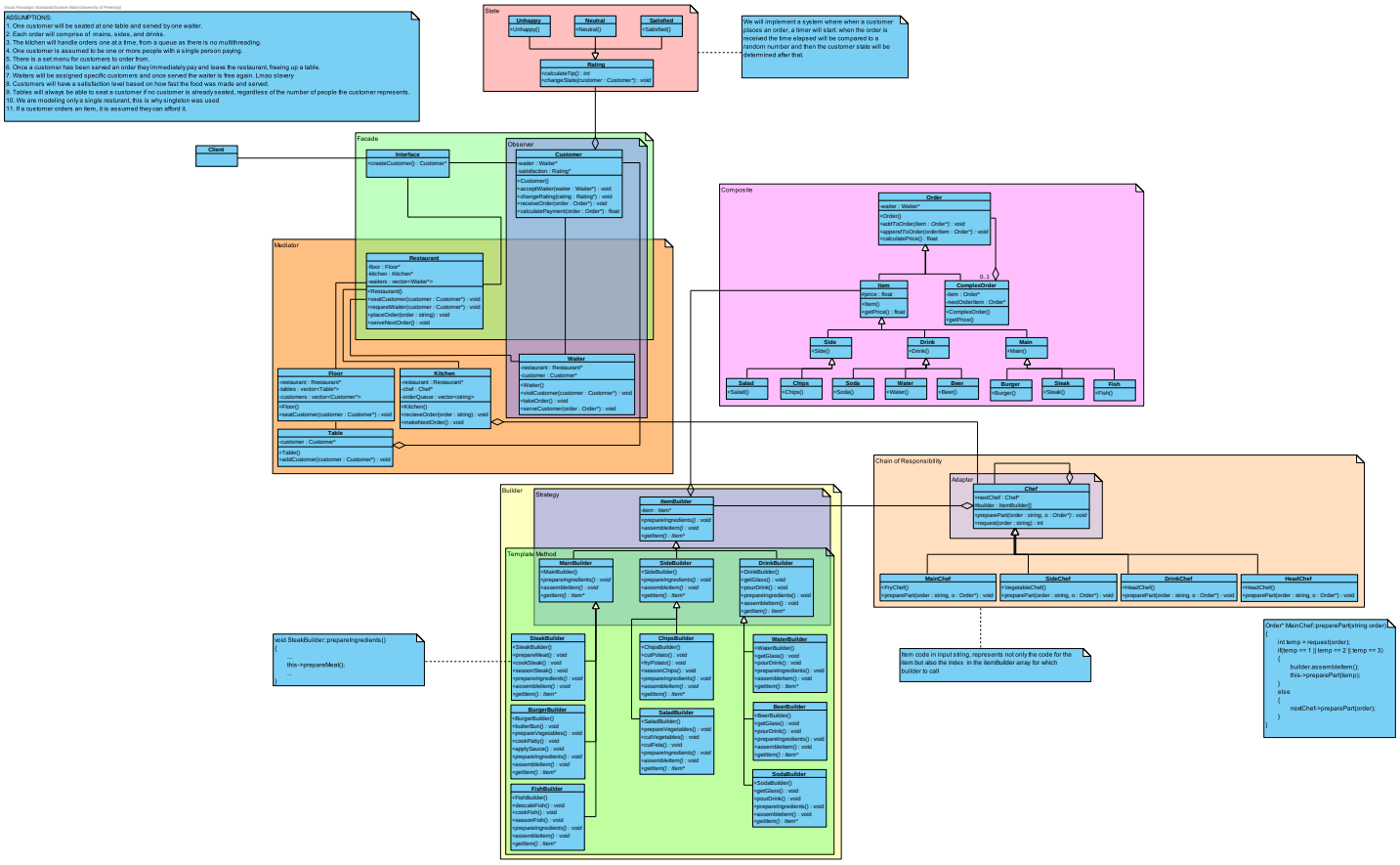
Visual Paradigm Standard(Dzianne Blain(University of Pretoria))
ASSUMPTIONS:
1. One customer will be seated at one table and served by one waiter.
2. Each order will comprise of mains, sides, and drinks.
3. The kitchen will handle orders one at a time, from a queue as there is no multithreading.
4. One customer is assumed to be one or more people with a single person paying.
5. There is a set menu for customers to order from.
6. Once a customer has been served an order they immediately pay and leave the restaurant, freeing up a table.
7. Waiters will be assigned specific customers and once served the waiter is free again, Lmao slavery
8. Customers will have a satisfaction level based on how fast the food was made and served.
9. Tables will always be able to seat a customer if no customer is already seated, regardless of the number of people the customer represents.
10. We are modeling only a single restuarant, this is why singleton was used
11. If a customer orders an item, it is assumed they can afford it.
State
Unhappy
+Unhappy()
Neutral
+Neutral()
Satisfied
+Satisfied()
Rating
+calculateTip() : int
+changeState(customer : Customer*) : void
We will implement a system where when a customer places an order, a timer will start, when the order is received the time elapsed will be compared to a random number and then the customer state will be determined after that.
Client
Facade
Interface
+createCustomer() : Customer*
Observer
Customer
-waiter : Waiter*
-satisfaction : Rating*
+Customer()
+acceptWaiter(waiter : Waiter*) : void
+changeRating(rating : Rating*) : void
+receiveOrder(order : Order*) : void
+calculatePayment(order : Order*) : float
Mediator
Restaurant
-floor : Floor*
-kitchen : Kitchen*
-waiters : vector<Waiter*>
+Restaurant()
+visitCustomer(customer : Customer*) : void
+requestWaiter(customer : Customer*) : void
+placeOrder(order : string) : void
+serveNextOrder() : void
Waiter
-restaurant : Restaurant*
-customer : Customer*
+Waiter()
+visitCustomer(customer : Customer*) : void
+takeOrder() : void
+serveCustomer(order : Order*) : void
Floor
-restaurant : Restaurant*
-tables : vector<Table*>
-customers : vector<Customer*>
+Floor()
+seatCustomer(customer : Customer*) : void
Kitchen
-restaurant : Restaurant*
-chef : Chef*
-orderQueue : vector<string>
+Kitchen()
+recieveOrder(order : string) : void
+makeNextOrder() : void
Table
-customer : Customer*
+Table()
+addCustomer(customer : Customer*) : void
Composite
Order
-waiter : Waiter*
+Order()
+addToOrder(item : Order*) : void
+appendToOrder(orderItem : Order*) : void
+calculatePrice() : float
Item
-price : float
+Item()
+getPrice() : float
0..1
ComplexOrder
-item : Order*
-nextOrderItem : Order*
+ComplexOrder()
+getPrice()
Side
+Side()
Drink
+Drink()
Main
+Main()
Salad
+Salad()
Chips
+Chips()
Soda
+Soda()
Water
+Water()
Beer
+Beer()
Burger
+Burger()
Steak
+Steak()
Fish
+Fish()
Builder
Strategy
ItemBuilder
-item : Item*
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
Template Method
MainBuilder
+MainBuilder()
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
SideBuilder
+SideBuilder()
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
DrinkBuilder
+DrinkBuilder()
+getGlass() : void
+pourDrink() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
SteakBuilder
+SteakBuilder()
+prepareMeat() : void
+cookSteak() : void
+seasonSteak() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
BurgerBuilder
+BurgerBuilder()
+buttonBun() : void
+prepareVegetables() : void
+cookPatty() : void
+applySauce() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
FishBuilder
+FishBuilder()
+descaleFish() : void
+cookFish() : void
+seasonFish() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
ChipsBuilder
+ChipsBuilder()
+cutPotato() : void
+fryPotato() : void
+seasonChips() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
SaladBuilder
+SaladBuilder()
+prepareVegetables() : void
+cutVegetables() : void
+mixFries() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
WaterBuilder
+WaterBuilder()
+getGlass() : void
+pourDrink() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
BeerBuilder
+BeerBuilder()
+getGlass() : void
+pourDrink() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
SodaBuilder
+SodaBuilder()
+getGlass() : void
+pourDrink() : void
+prepareIngredients() : void
+assembleItem() : void
+getItem() : Item*
void SteakBuilder::prepareIngredients()
{
...
this->prepareMeat();
...
}
Chain of Responsibility
Adapter
Chef
+nextChef : Chef*
-builder : ItemBuilder[]
+prepareFor(order : string, o : Order*) : void
+request(order : string) : int
MainChef
+FryChef()
+prepareFor(order : string, o : Order*) : void
SideChef
+VegetableChef()
+prepareFor(order : string, o : Order*) : void
DrinkChef
+HeadChef()
+prepareFor(order : string, o : Order*) : void
HeadChef
+HeadChef()
+prepareFor(order : string, o : Order*) : void
Item code in input string, represents not only the code for the item but also the index in the itemBuilder array for which builder to call
Order* MainChef::prepareFor(string order)
{
int temp = request(order);
if(temp == 1 || temp == 2 || temp == 3)
{
builder.assembleItem();
this->prepareFor(temp);
}
else
{
nextChef->prepareFor(order);
}
}