

**EGE UNIVERSITY**

**FACULTY OF ENGINEERING**

**DEPARTMENT OF COMPUTER ENGINEERING**

**SOFTWARE ANALYSIS AND DESIGN**

**2022–2023 SPRING SEMESTER**

**PROJECT**

**DELIVERY DATE**

**3/07/2023**

**PREPARED BY**

**05200000096-Uğur Alp Balta**

**Use Case : Handle Return**

**Scope:** Point of Sale System

**Level:** Cashier Goal

**Primary Actor:** Cashier

**Stakeholders and Interests:**

**-Customer:** Person who return products from a previous purchase.

-**Cashier:** Person who wants to update the POS system for.

**Preconditions: -**

**Success Guarantee (or Post Condition) :** Customer successfully returns products.

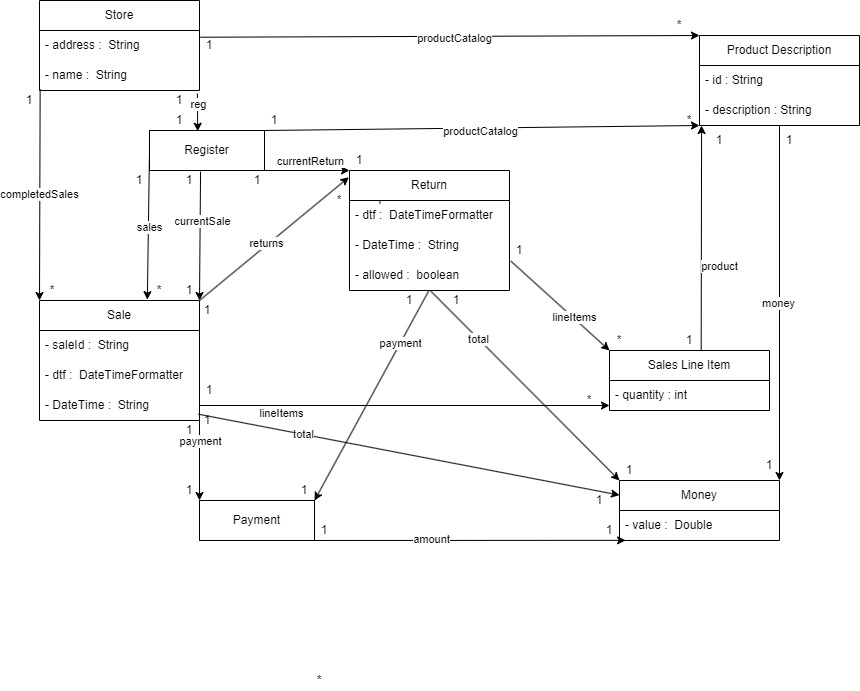
**Main Success Scenario(or Basic Flow):**

|  |  |
| --- | --- |
| 1. The customer arrives at POS checkout with receipt from a previous purchase and provides said receipt to the Cashier. |  |
| 2. The cashier issues a Handle Return process |  |
|  | 3. The system requests for the sale id. |
| 4. The cashier provides the sale id from the receipt. |  |
|  | 5. If the identifier is valid, the system asks for idendifier and quantity of a retuning item. |
| 6. The cashier provides the product id and quantity from the receipt. |  |
|  | 7. If the item Id is valid and the items and quantity of the item is correct corresponding to the sale, the system saves the item. Then ask if there are more items or not. |
| 8. The Cashier answers in the affirmative or in the negative |  |
|  | 9. If affitmative system goes to step 6 otherwise the system prints all the products that should be returned from said sale and the calculated total amount to be paid to the customer. Then asks for a return confirmation. |
| 10. The Cashier answers in the affirmative or in the negative |  |
|  | 11. If Affirmative the system asks for payment confirmation. |
| 12. The Cashier answers in the affirmative or in the negative |  |
|  | 13. If Affirmative, the system saves the Return, prints the receipt then exits |

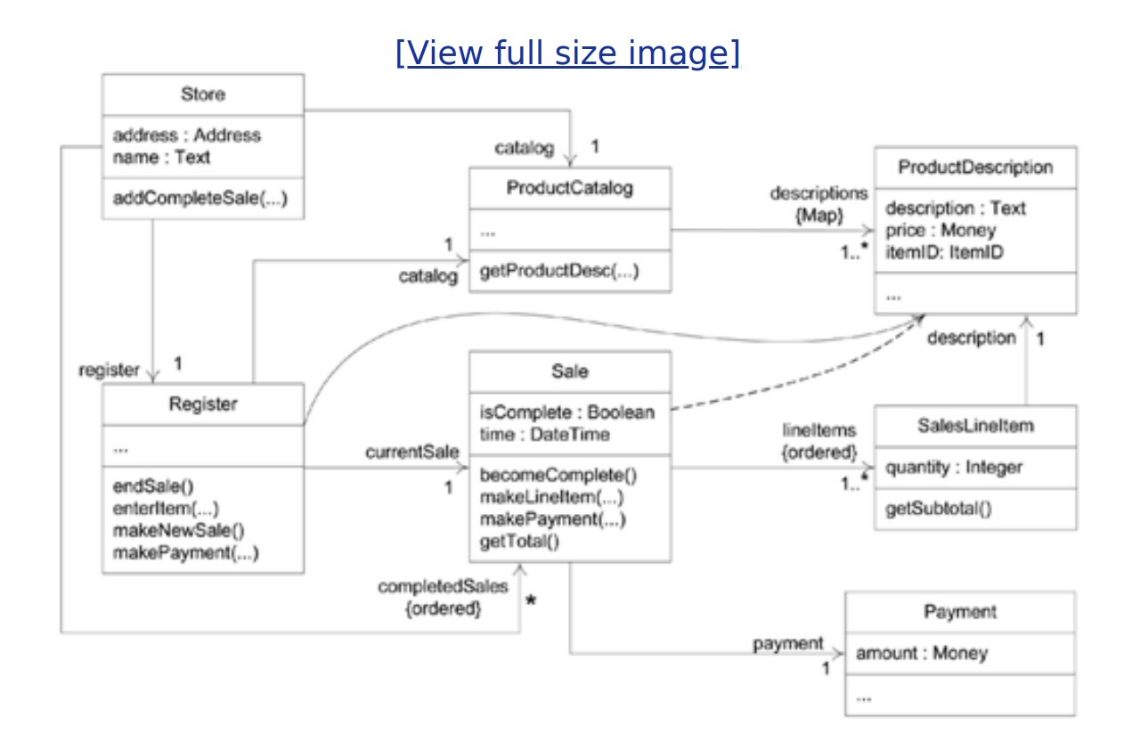
It should be noted that in this design a customer can return one or many products from their previous purchase. So they can return only some portion of the poducts in varying quantites. Also since only a portion of the products returned this time. So each sale contains a returns list for containing information and dates about different returns from the same sale.

**Domain Model:**

This domain model shows every containment relationship between classes outside of the interface.

****

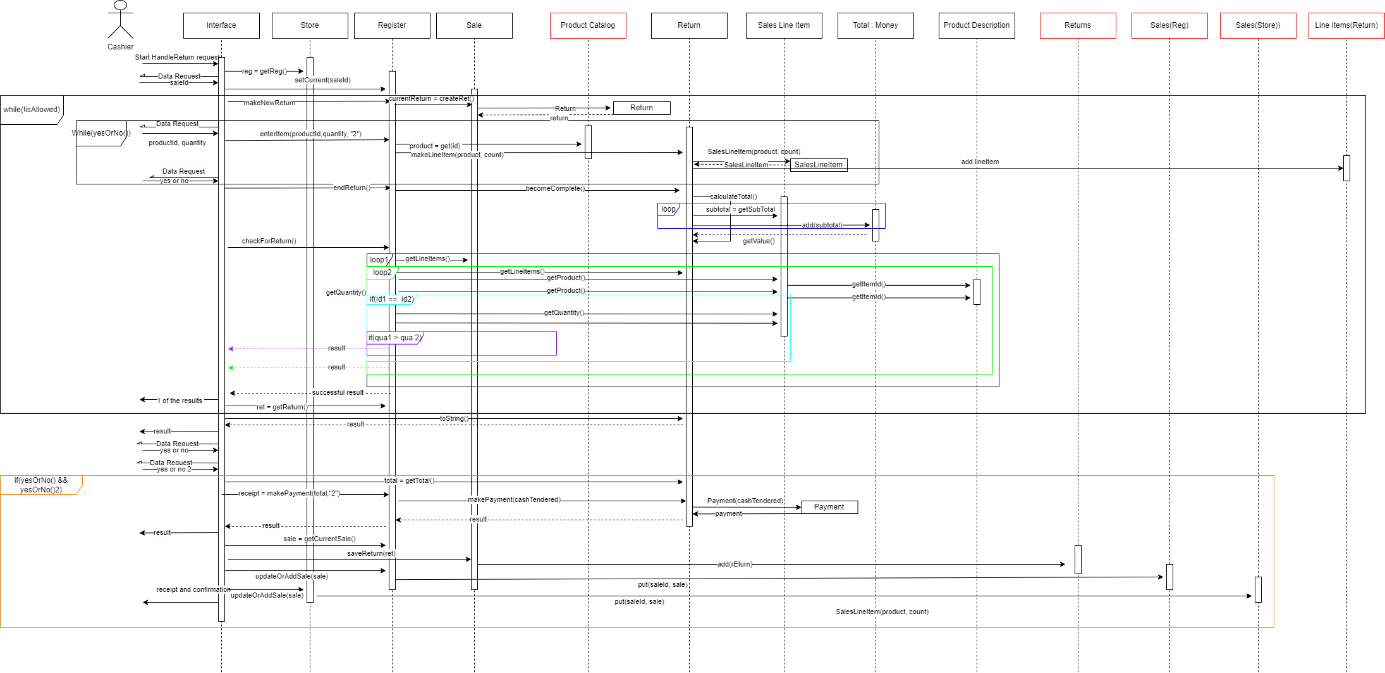
**Old Version:**



**Sequence Diagram : Handle Return**

The boxes which represtns hashmaps or lists instead of classes are marked with red to avoid any confusion.

This sequence diagram shows almost every method execution in the code. Since it is incredibly complex due to complex nature of this system some colorations used to make distinguishing between the lines easier.



**Explanation in Terms of GRASP Patterns**

**Controller:** The Register class controls the sale and return processes, manages the current sale and return objects, and interacts with other classes such as "ProductDescription," "Sale," "Return," and "Payment." It encapsulates the business logic related to these processes, including handling line item entries, calculating totals, and processing payments. But it is not alone in this;

The Interface class handles user interactions and controls the flow of operations in response to user commands. It provides methods for processing sales, handling returns, displaying help information, and managing the program's execution.

This dual controllership happened because of the system given in the book, since in the book flow of operations was controlled by JFrames (Interface calls the Register functions one by one as seen in the example I attached) I had to do the same with my interface class.

If I wasn’t going for the “by the book” approach I could’ve handled the handleReturn method inside of the register class by giving the retuning items inside of the in bulk similar to library system..

**metin, ekran görüntüsü, çizgi, yazı tipi içeren bir resim

Açıklama otomatik olarak oluşturulduExample:**

**Creator:** The Return class acts as a creator by creating and storing SalesLineItem and Payment objects within it. Additionally, the Sale class creates a new Return object when a return is initiated. Both classes exhibit the Creator pattern by encapsulating the creation of related objects.

**Information Expert:** The Return class encapsulates information and behavior related to the return process. It manages a list of SalesLineItems, calculates the total refund amount, handles the payment for the return, and provides a formatted string representation of the return details. It also interacts with the "SalesLineItem" class to calculate subtotals. But since without the information about previous sale (for comparison) couldn’t be Access without Sale class has some properties that could be considered information expert as well. (Although the answer is clearly Return)

**High Cohesion:** The system exhibits high cohesion as each class focuses on a specific responsibility. The Money class represents a monetary value and provides methods for manipulating it. The ProductDescription"class represents information about a product and manages its attributes. The SalesLineItem class represents a line item in a sale, calculates subtotals, and provides a formatted string representation. The Payment class represents a payment made during a sale or return. The Return class manages the return process, including line items, total calculation, and payment handling. The Sale class manages the sale process, including line items, total calculation, and payment handling. The "Store" class represents a store and manages the product catalog and sales history. The "Register" class handles the sale and return processes, manages the current sale and return objects, and interacts with the product catalog and sales history. The "Interface" class acts as the user interface, handling user interactions and controlling the flow of operations.

**Low Coupling:** The system exhibits low coupling as the classes interact with each other through references received as parameters or obtained from other objects. The Interface class interacts with the Store class to access the Register object. The Store class interacts with the Register class and the ProductDescription class. The Register class interacts with the ProductDescription class, the Sale class, the Return class, and the Payment class. These interactions are based on the specific needs of the operations being performed.

**Code Implementation:**

Note that as said in the course we assume all the necessary information is already pulled from the database into hashmaps (in the unincluded main method) and the System is already complete information-wise. Also since the code must be working(if a main test method is written) and there wasn’t any full working code in the book, I found(Since this is an update homework), fixed and updated (In scope of handleReturn. Also added things like Interface object, Return object, Store Object exc. ) another processSale code. Also kept and updated processSale so handleReturn could be usable by creating a sale first. While writing the code, I prioritze the book’s design since it was stated that I need to update instead of writing a seemingly slightly better one in terms of GRASP patterns. Code that aren’t used in the handleReturn process marked with “unnecessary” they are there for either fort he processSale process, implementing the code without a database or future proofing,. I also included full code(again, without the main method) as a Java project inside of the file.

**Interface: (**Borrowed from the library system**):**

public class Interface {  
 Store store;  
 Register reg;  
 public Interface(Store store){  
 this.store = store;  
 }  
 private static Scanner *scanner* = new Scanner(System.*in*);  
 // Method to get a valid command from the user  
 public static String getCommand() {  
 System.*out*.print("Enter a command: ");  
 String command = *scanner*.nextLine();  
  
 // Validate the command  
 if (!command.equals("PROCESS\_SALE") && !command.equals("HANDLE\_RETURN") && !command.equals("HELP") && !command.equals("EXIT")) {  
 System.*out*.println("Invalid command. Please try again.");  
 return *getCommand*(); // Recursive call to get a valid command  
 }  
 return command;  
 }  
 // Method to get a token from the user  
 public static String getToken(String message) {  
 System.*out*.print(message + ": ");  
 return *scanner*.nextLine();  
 }  
 // Method to get a yes or no answer from the user  
 public static boolean yesOrNo(String message) {  
 System.*out*.print(message + " (y/n): ");  
 String input = *scanner*.nextLine().toLowerCase();  
 return input.equals("y") || input.equals("yes");  
 }  
 // Method to display available commands  
 public void help() {  
 System.*out*.println("Welcome to the program!");  
 System.*out*.println("Available commands:");  
 System.*out*.println("- 'PROCESS\_SALE': Process a sale");  
 System.*out*.println("- 'HANDLE\_RETURN': Handle a return");  
 System.*out*.println("- 'HELP': Show available commands");  
 System.*out*.println("- 'EXIT': Exit the program");  
 }  
 // Method to process a sale  
 public void processSale(){ // unnecessary  
 reg = store.getReg();  
 reg.makeNewSale();  
 // Get product details and add them to the sale  
 do {  
 String productId = *getToken*("Please enter product id");  
 int quantity = Integer.*parseInt*(*getToken*("Please enter product quantity"));  
 reg.enterItem(productId, quantity, "2");  
 } while (*yesOrNo*("Add more products?"));  
  
 reg.endSale();  
 Sale sale = reg.getCurrentSale();  
 Money total = sale.getTotal();  
 System.*out*.println("Payment that should be taken from the customer is: " + total);  
  
 // Check if the payment has occurred  
 if(*yesOrNo*("Did the payment happen?")){  
 String receipt = reg.makePayment(total, "1");  
 //Added so in continuous usage of register it stays updated  
 reg.updateOrAddSale(sale);  
 store.updateOrAddSale(sale);  
 System.*out*.println(receipt);  
 System.*out*.println("Sale process has been completed and saved.");  
 }  
 }  
 // Method to handle a return  
 public void handleReturn(){  
 boolean isAllowed = false;  
 Return ret;  
 reg = store.getReg();  
 String saleId = *getToken*("Please enter sale id");  
 reg.setCurrent(saleId);  
 int counter = 0;  
 do {  
 if(counter != 0){  
 if(!*yesOrNo*("The return wasn't processable. Try again?")){  
 return;  
 }  
 }  
 reg.makeNewReturn();  
 // Get product details and add them to the return  
 do {  
 String productId = *getToken*("Please enter product id");  
 int quantity = Integer.*parseInt*(*getToken*("Please enter product quantity"));  
 reg.enterItem(productId, quantity, "1");  
 } while (*yesOrNo*("Add more products?"));  
  
 reg.endReturn();  
 System.*out*.println(reg.checkForReturn());  
 ret = reg.getReturn();  
  
 if(ret.isAllowed()){  
 isAllowed = true;  
 }  
 counter++;  
 } while(!isAllowed);  
 // Check if the payment has occurred  
 System.*out*.println(ret);  
 Money total = ret.getTotal();  
 if(*yesOrNo*("Did the customer provide every item returning?")){  
 if(*yesOrNo*("Did the payment happen?")){  
 String receipt = reg.makePayment(total, "2");  
 Sale sale = reg.getCurrentSale();  
 sale.saveReturn(ret);  
 //Added so in continuous usage of register it stays updated  
 reg.updateOrAddSale(sale);  
 store.updateOrAddSale(sale);  
 System.*out*.println(receipt);  
 System.*out*.println("Return process has been completed and saved.");  
 }  
 }  
  
 }  
 // Method to start the interface and process user commands  
 public void process() {  
 String command;  
 help();  
  
 // Continue processing commands until the user chooses to exit  
 while (!(command = *getCommand*()).equals("EXIT")) {  
 switch (command) {  
 case "PROCESS\_SALE" -> processSale(); // unnecessary  
 case "HANDLE\_RETURN" -> handleReturn();  
 case "HELP" -> help();  
 }  
 }  
 }  
}

**Money:**

public class Money {  
 private double value;  
  
 //Initialize zero valued money  
 public Money(){  
 value = 0.0;  
 }  
 //Initialize money with the given amount  
 public Money(double value){  
 this.value = value;  
 }  
 public double getValue(){  
 return value;  
 }  
 public void setValue(double value){  
 this.value = value;  
 }  
 //Add some double amount to the money  
 public void add(double amount){  
 value += amount;  
 }  
 //Multiply the money with the given quantity variable  
 public double multiply(int quantity) {  
  
 return value \* quantity;  
 }  
}

**Payment:**

public class Payment {  
 private Money amount;  
  
 //Payment is a type of money exchange between customer and store  
 public Payment(Money cashTendered) {  
  
 amount = cashTendered;  
 }  
 public void setAmount(Money amount){ // unnecessary  
 this.amount = amount;  
 }   
 public Money getAmount() { // unnecessary  
  
 return amount;  
 }  
}

**Product Description:**

public class ProductDescription {  
  
 private int id;  
 private Money money;  
 private String description;  
  
 // Represents individual products with their information  
 public ProductDescription(int id, Money money, String description) {  
 this.id = id;  
 this.money = money;  
 this.description = description;  
 }  
  
 public int getItemID() {  
 return id;  
 }  
  
 // Set the price of the product  
 public void setPrice(Double price){ // unnecessary  
 money.setValue(price);  
 }  
  
 public Money getPrice() {  
 return money;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 // Override the toString() method to provide a custom string representation of the object  
 @Override  
 public String toString(){  
 return String.*format*("%1$"+14+ "s", getDescription()+" → ");  
 }  
}

**SalesLineItem:**

public class SalesLineItem {  
 private int quantity;  
 private ProductDescription product;  
  
 public SalesLineItem(ProductDescription product, int quantity) {  
 this.product = product;  
 this.quantity = quantity;  
 }  
  
 public ProductDescription getProduct() {  
 return product;  
 }  
  
 public int getQuantity() {  
 return quantity;  
 }  
  
 public Double getSubtotal() {  
 // Calculate the subtotal by multiplying the product price with the quantity  
 return product.getPrice().multiply(quantity);  
 }  
  
 @Override  
 public String toString() {  
 String str = product.toString() + " ";  
 String str1 = String.*format*("%1$" + 3 + "s", quantity);  
 String str2 = String.*format*("%1$" + 5 + "s", " x ");  
 String str3 = String.*format*("%1$" + 5 + "s", product.getPrice().getValue());  
  
 return str + str1 + str2 + str3 + " = " + getSubtotal() + "$";  
 }  
}

**Return:** (A similar object to sale that holds a spesific return for said sale)

public class Return {  
 private List<SalesLineItem> lineItems;  
 private DateTimeFormatter dtf;  
 private Payment payment;  
 private Money total;  
 private boolean allowed;  
  
 private String DateTime;  
 public Return(){  
  
 lineItems = new ArrayList<SalesLineItem>();  
 dtf = DateTimeFormatter.*ofPattern*("yyyy/MM/dd HH:mm");  
 //Initialize empty money object to hold the total of the return  
 total = new Money();  
 }  
 public boolean isAllowed() {  
 return allowed;  
 }  
  
 public void setAllowed(boolean allowed) {  
 this.allowed = allowed;  
 }  
 public List<SalesLineItem> getLineItems() {  
 return lineItems;  
 }  
 public String getDateTime() {  
 return DateTime;  
 }  
 public Money getTotal() {  
 return total;  
 }  
  
 public Payment getPayment() {// unnecessary  
 return payment;  
 }   
 public void becomeComplete() {  
 //Calculate the total amount when the return is complete  
 DateTime = dtf.format(LocalDateTime.*now*());  
 calculateTotal();  
 }  
 public void makeLineItem(ProductDescription product, int quantity) {  
 //Add new item and its quantity to the return  
 lineItems.add(new SalesLineItem(product, quantity));  
 }  
 public Double calculateTotal() {  
 Double subtotal;  
 //Find the subtotal of each SaleLineItem  
 for (SalesLineItem lineItem : lineItems) {  
 //Multiplied by the quantity and product's price  
 subtotal = lineItem.getSubtotal();  
 total.add(subtotal);  
 }  
 return total.getValue();  
 }  
 public String makePayment(Money cashTendered) {  
 //When the return is complete , customer pays the amount  
 payment = new Payment(cashTendered);  
 //After payment is received, the return is recorded to a text file  
 return this.toString();  
 }  
  
 @Override  
 public String toString() {  
 String receipt = "\n ➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤ \n\n";  
 //Details of the return  
 for (SalesLineItem item : lineItems) {  
 receipt += "✯ " + item.toString() + "\n";  
 }  
 receipt += "✎ Total: " + Double.*toString*(total.getValue()) + " $ \n";  
 //Timestamp of the sale  
 receipt += "⌛ " + DateTime + "\n";  
 return receipt;  
  
 }  
}

**Sale:**

public class Sale {  
 private String saleId;  
 private List<SalesLineItem> lineItems;  
  
 private List<Return> Returns; // Consider renaming this field to "returns" (lowercase "r") for naming consistency  
  
 private DateTimeFormatter dtf;  
  
 private Payment payment;  
 private Money total;  
  
 private String DateTime; // Consider renaming this field to "dateTime" (lowercase "d") for naming consistency  
  
 public Sale(String saleID){  
 //Initialize a list to hold the items of the sale  
 this.saleId = saleID;  
 lineItems = new ArrayList<SalesLineItem>();  
 dtf = DateTimeFormatter.*ofPattern*("yyyy/MM/dd HH:mm");  
 //Initialize empty money object to Hold the total of the sale  
 total = new Money();  
  
 }  
 public Money getTotal() { // unnecessary  
 return total;  
 }  
 public String getSaleId() {  
 return saleId;  
 }  
 public Payment getPayment() { // unnecessary  
 return payment;  
 }  
 public List<SalesLineItem> getLineItems() {  
 return lineItems;  
 }  
 public String getDateTime() { // unnecessary  
 return DateTime;  
 }  
 public List<Return> getReturns() { // unnecessary  
 return Returns;  
 }  
 public void becomeComplete() {  
 //Calculate the total amount when the sale is complete  
 DateTime = dtf.format(LocalDateTime.*now*());  
 calculateTotal();  
 }  
 public void makeLineItem(ProductDescription product, int quantity) {  
 //Add new item and its quantity to the sale  
 lineItems.add(new SalesLineItem(product, quantity));  
 }  
  
 public void saveReturn(Return rEturn){  
 Returns.add(rEturn);  
 }  
 public Double calculateTotal() {  
 Double subtotal;  
 //Find the subtotal of each SaleLineItem  
 for (SalesLineItem lineItem : lineItems) {  
 //Multiplied by the quantity and product's price  
 subtotal = lineItem.getSubtotal();  
 total.add(subtotal);  
 }  
 return total.getValue();  
 }  
 public String makePayment(Money cashTendered) {  
 //when the sale is complete , customer pays the amount  
 payment = new Payment(cashTendered);  
 //After payment is received, the sale is recorded to a text file  
 return this.toString();  
 }  
 public Return createRet(){  
 return new Return();  
 }  
  
 @Override  
 public String toString() {  
 String receipt = "\n ➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤➤ \n\n";  
 receipt += "Sale Id" + saleId +"\n";  
 //Details of the sale  
 for (SalesLineItem item : lineItems) {  
 receipt += "✯ " + item.toString() + "\n";  
 }  
 receipt += "✎ Total: " + Double.*toString*(total.getValue()) + " $ \n";  
 //Timestamp of the sale  
 receipt += "⌛ " + DateTime + "\n";  
 return receipt;  
  
 }  
  
}

**Store:**

public class Store {  
 private String address;  
 private String name;  
 private Map<String, Sale > completedSales;  
 private Register reg;  
 private Map<String, ProductDescription> productCatalog;  
 public Store(String name, String address){  
 productCatalog = new HashMap<String, ProductDescription>();  
 this.name = name;  
 this.address = address;  
 }  
 public Register getReg() {  
 return reg;  
 }  
 public void updateOrAddSale(Sale sale) {  
 completedSales.put(sale.getSaleId(), sale);  
 }  
 // The method below is added so code could run without a database  
 public void addProduct(String productId, ProductDescription product) { // unnecessary  
 productCatalog.put(productId, product);  
 }  
 public Register create(){ // unnecessary  
 reg = new Register(productCatalog, completedSales);  
 return reg;  
 }  
}

**THANKS FOR READING!**