Inventory System Business Requirements

**Background:**

If you have ever had occasion to shop in a supermarket or department store, you should be familiar with operation of a Point of Sale system

Each store keeps track of each Item that it sells. When a Customer purchases an Item, the Cashier scans the barcode to identify the Item. This allows the system to look up the current price that is being charged for the item (this price may vary as prices increase and sometimes be lower if the Store is running a Sale). The quantity in stock is decreased and the Customer is charged for the price \* quantity of units he is purchasing. When the Cashier has completed scanning all the Items, the Cashier often asks the Customer to identify himself (by providing his phone number) and the Cashier accepts payment for the Purchase in the form of Cash, Credit Card, Debit Card, EBT Card or a combination of these forms. Note: not all purchases can be paid for using an EBT card. The Customer then receives a Receipt that lists each of the Items he has purchased, the total amount due, the amount paid and the method of Payment. Some stores maintain information about their steady Customers including their phone numbers, each of their Purchases and Credit Card and/or EBT Card information. The Store can use this information to analyze trends, enabling it to maintain inventory of popular Items, run Sales at opportune times etc.

The Store buys the Items from Vendors. The Store orders the Items and pays for them once they have been Received.

**Specifications**:

* Design and implement a Point of Sale system that will manage a Store’s inventory purchases and sales
* Use the OO approach to design and implement the system.
* To get you started, a set of collaborative classes has been identified and laid out in the UML provided but feel free to modify the design if you determine that the specifications warrant modifications.
* Include data validation within constructors and setters
* Implement meaningful Exception classes and determining which should extend RuntimeException versus Exception
* Have each class implement Serializable so that the entire POS can be stored and restored from a disk file
* Designate some getters with protected accessibility when necessary to protect access to private data members.
* Only implement setters as the situation warrants; some classes should not support any setters, while others should support setters for certain attributes and not for others.
* Implement a user friendly interface that will allow Employees to manage inventory (place PurchaseOrders, process Shipments of Items from Vendors), Customers to buy Items, Cashiers to complete sales to Customers, Employees to analyze the financial state of the Store income and expenses and customer base

Specifications for some of the Classes have been outlined below, following the CRC approach (Class, Responsibilities and Collaboration approach).

To further assist I have included sample methods. Feel free to add additional methods or modify the methods keeping in mind that you must ensure that the class continues to meet the specifications outlined.

Note: Classes that have a “has a” relationship with another class can either store the primary key that identifies an instance of the other class or store a direct reference to the instance. Many of the classes that are outlined below are designed to store the primary key however, you are encouraged to store a direct reference to the instance whenever you deem that it might facilitate subsequent processing. Example: a SalesOrder may be associated with a Customer. The SalesOrder instance can either store the CustomerID of the Customer or a direct reference to the Customer.

|  |  |  |
| --- | --- | --- |
| Class | Item |  |
| Superclass |  |  |
| Subclass | None |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of an Item | Allow certain attributes to be modified and provide indirect access to all attributes |  |
| verifyUPC(String upc)  *class method* | A fruit or veggie must have a code of either 4 or 5 digits. Note: A 5 digit code indicates organically grown fruit or veggie  UPC can be either 8, 12 or 13 digits long.  To verify a UPC, verify that the length is either 8,12 or 13. Note: Doesn’t verify fruit or veggie codes  Calculate the value of the check digit (appears as the rightmost digit of the code) as follows:  Omit the check digit, number the positions from the right, multiply each value in an odd position by 3 and the even data digits by 1. Add up the result.  Subtract the sum from the nearest multiple of 10 that is higher than the sum.  This value should equal the check digit that appears in the UPC.  Example:  UPC bar code **7**2**4**3**2**7**0**0**0**0**2**3 The check digit is 3  Let’s verify 3 \* ( 7 + 4 + 2 + 0 + 0 + 2 ) + (2 + 3 + 7 + 0 + 0 )  45 + 12 = 57  Next multiple of 10 = 60 , 60 – 52 = 3 matches | ItemType  ItemDiscount  ItemDescriptionComparator  UnitMeasure  Stack |
| public Item(String upc, String desc, Double price, String type, String unitMeasure, int vendorid) { }    public void addDiscount (LocalDate start, LocalDate end, int qtyLimit, Double discountPrice) {}  public void addDiscount(ItemDiscount theDiscount) { }    public Double getDiscountPrice(LocalDate currentDate) {  public Item clone() {}    public String toString() {}    getters  setters  equals()  compareTo() | constructor  Add an ItemDiscount price to the discounts that have been applied to this Item in the past. An ItemDiscount can’t be applied to an Item more than once in a specific month.  Return the latest discount price(if any exists) that has been applied to the Item if the currentDate is within the time range of the discount.  Instantiate a deep copy which includes only the latest discount offered on this Item  Do not allow the UPC to be modified. The UPC is a primary key |  |

|  |  |  |
| --- | --- | --- |
| Class | Person |  |
| Superclass |  |  |
| Subclass | Cashier, Customer |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Person  Maintain a *class* attribute , ***lastID*** | Provide constructor, getters and setters  Assign a unique ID to each instance of a Person  The ID should not be modified since it is a primary key | Address |
| public Person(String firstName, String lastName, Address address,String phoneNumber,char gender)  toString()  equals()  compareTo() | constructor |  |

|  |  |  |
| --- | --- | --- |
| Class | Address |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of an Address | Provide constructor, getters  If you wish to reference the same Address more than once consider assigning a unique AddressID and maintaining a class attribute, lastAddressID | USState |
| Address(String street, String city, String State, String zip)  Address(Address)  equals()  compareTo()  toStreet()  clone() | Constructor - ensure that the state is a valid state.  Return deep copy of Address |  |

|  |  |  |
| --- | --- | --- |
| Class | Cashier |  |
| Superclass | Person |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Cashier | Provide constructors, getters and setters (if applicable) |  |

|  |  |  |
| --- | --- | --- |
| Class | Customer |  |
| Superclass | Person |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Customer | Provide constructors, getters and setters | SalesOrder  SalesOrderDetail  Payment  Stack  ArrayList  Payment  CardType |
| public Customer(String lastName, String firstName,String phoneNumber, Address address,char gender)  public void addSale(SalesOrder theSale) {}    public void completeSale(SalesOrder)  public void addPayment(Payment thePayment) {    protected SalesOrder getLatestSalesOrder() { }  Double processReturn(CustomerReturn)  equals()  compareTo()  toString() | Constructor  Add SalesOrder to collection of SalesOrders, can keep additional reference to the SalesOrder to ease searching and retrieval  Verify that this SalesOrder isn’t already in the Customer collection of SalesOrders. Verify that this SalesOrder has been completed. If yes, add this SalesOrder to the Customer’s collection of SalesOrders. Add the ‘total’ of the SalesOrder to the Customer ‘balance’  Add the Payment reference to the list of Payments  Modify the balance to reflect the Payment  Return direct reference to the latest SalesOrder  Search for the SalesOrderDetail that corresponds to this Return, calculate the refund amount, based on what the customer paid for this Item when purchased |  |

|  |  |  |
| --- | --- | --- |
| Class | CustomerReturn |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a returned item | Provide constructors, getters |  |
| CustomerReturn(long salesOrderID, String upc, int qtyReturned, String reason)  equals()  compareTo()  toString() | Note : an Item can’t be returned by a Customer unless it has been purchased by that Customer. |  |

|  |  |  |
| --- | --- | --- |
| Class | SalesOrder |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a SalesOrder  Maintain a class attribute “lastSalesOrderID” | Provide constructors, getters and setters  Each SalesOrder is assigned a unique SalesOrderID | SalesOrderDetail  Payment  PriorityQueue based on ItemTypeComparator |
| public SalesOrder(Long customerID,int cashierID,ArrayList<SalesOrderDetail> details) { }  public SalesOrder (Long customerID, int cashierID) { }    **protected** Payment getPayment() {}    public void addDetail (SalesOrderDetail detail ) {    public void addDetails(ArrayList<SalesOrderDetail>  public void removeDetail(SalesOrderDetail)  public void applyDiscount(double minimumPurchase)  public void makePayment(Payment)  public String printReceipt()  toString() | Constructor – the CustomerID can be null if the SalesOrder isn’t associated with a regular Customer  Can modify the constructors to accept a reference to a Customer instance if you modify the class to reference a specific Customer.  Return reference to Payment  If the SalesOrder hasn’t been completely processed  Add a SalesOrderDetail to list of SalesOrderDetails, modify the totalSale based on data in the SalesOrderDetail. Modify the EBTTotal if the SalesOrderDetail represents an Item that is covered by the EBT program. If the Item is not on sale, add the total of this sale to ‘totalNonDiscount’ of this SalesOrder  If the SalesOrder hasn’t been completed  Verify that the SalesOrderDetail is part of this SalesOrder  Reverse the entry by adjusting the SalesOrder totals  If total of non sale items >= minimum purchase, for each SalesOrderDetail that represents an Item currently on sale adjust the total of the SalesOrder accordingly so that it reflects the sale price and not the regular price for this Item  Verify that Payment hasn’t already been made for this SalesOrder. Verify the PaymentDetails to ensure that EBT card is used to pay amount <= EBTTotal |  |
| Class | ItemTypeComparator |  |
| Interface | Comparator |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Compare two SalesOrderDetail instances based on the ItemType of the SalesOrderDetail |  | SalesOrderDetail  ItemType |

|  |  |  |
| --- | --- | --- |
| Class | SalesOrderDetail |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a SalesOrderDetail | Provide constructor, getters | ItemType |
| public SalesOrderDetail(Long orderID,String upc,int qty, double price,Double discountPrice, boolean onSale, boolean isEBT,ItemType type) { }  public String toString() {}  equals()  compareTo() |  |  |

|  |  |  |
| --- | --- | --- |
| Class | Payment |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Payment  Maintain the class attribute, ‘lastPaymentID’  public Payment(Long orderID, PaymentType paymentType)  public void addDetail (double amount, String method) { }    public void addDetail(double amount, String method,String cardID) { }    public void addDetail(PaymentDetail detail)  public String toString() | Provide constructor, getters  Each Payment should be assigned a unique PaymentID  A Payment can either be a Payable or Receivable  A Payment is a Receivable if it comes from a Customer who is paying for a SalesOrder  A Payment is Payable if it is being paid to a Vendor to cover the costs of a PurchaseOrder  Instantiate a PaymentDetail instance, add to the list of details and modify the Payment’s ‘total’ accordingly | PaymentDetail  PaymentType  ArrayList |

|  |  |  |
| --- | --- | --- |
| Class | PaymentDetail |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a PaymentDetail  public PaymentDetail(Long paymentID, String method, double amount)    public PaymentDetail(Long paymentID, String method, double dollarAmount,String cardID)  public String toString() | Provide constructor, getters  Instantiate a PaymentDetail instance.  If the PaymentMethod is a Credit card or Debit card or EBT card must provide a cardID | PaymentMethod |

|  |  |  |
| --- | --- | --- |
| Class | Vendor |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Vendor  Maintain the class attribute  ‘lastVendorID’ | Provide constructor, getters and setters (where appropriate | Address  PurchaseOrder  Stack  HashMap |
| public Vendor(String vendorName, Address address,String phoneNumber)  public void addPurchaseOrder(PurchaseOrder order)  protected PurchaseOrder getPurchaseOrder(Long poID)    public Double getTotalOwed()    public String toString() | Instantiate an instance of a Vendor  Add the PurchaseOrder to the collection of PurchaseOrders  Return direct reference to PurchaseOrder that has PurchaseOrderID equal to poID  Calculate total amount still owed to this Vendor for PurchaseOrders that have not been paid. |  |

|  |  |  |
| --- | --- | --- |
| Class | PurchaseOrder |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a PurchaseOrder  Maintain the class attribute  ‘lastOrderID’ | Provide constructor, getters | PurchaseOrderDetail  Payment  HashMap |
| public PurchaseOrder(int vendorID) {    public void addOrderDetails(PurchaseOrderDetail details)    public void addOrderDetails(String upc,int qtyOrdered)    public void receivedItem(String upc, int qty, double unitCost)    public void makePayment(Payment thePayment)    equals()  compareTo()  public String toString() | Instantiate a PurchaseOrder, the purchaseDate is the current date  Add PurchaseOrderDetails to the collection of PurchaseOrderDetails  Instantiate a PurchaseOrderDetail and add it to the collection of PurchaseOrderDetails  Verify that this Item has been ordered  If yes, modify the PurchaseOrderDetail to reflect the date the Item has been received, the quantity received and the unit cost. Update the totalOrderAmount to reflect the amount that must be paid for the Item received.  Add the Payment to the collection of Payments  Modify the totalOrderAmount to reflect the amount of the Payment to the vendor for this PurchaseOrder |  |

|  |  |  |
| --- | --- | --- |
| Class | Store |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of a Store | Provide constructor, getters  Include methods that provide information such as which Vendor supplies the most products, which customer has the largest balance, which customer generated the most income, which product sold the most often, which products have been offered at sale price | Address  Inventory  Vendor  Person  PurchaseOrder  SalesOrder  Payment  HashMap  TreeMap  SalesOrderDetail  PurchaseOrderDetail  Customer  Item |
| public Store(String name, Address address, String phoneNumber)    public void storeData(){ }    public void addItem(Item theItem)    public void addVendor(Vendor theVendor)    public void receiveItemShipment(Long poID, String upc, int qty, Double unitCost)    public void receiveShipment(Long poID, ArrayList<ReceivedItem> items)    public Long setUpPurchaseOrder(Integer vendorID)    public Long setUpPurchaseOrder(int vendorID, ArrayList<ItemOrder> items)            public void addItemToPOFromVendor(Long poID, String upc,Integer qtyOrdered)          public void receivePayment(Long orderID,ArrayList<PaymentDetail>details)        public void makePayment(int vendorID, long purchaseOrderID,double amount)          public void setRestockLevel(String upc, int level) {    public boolean hasItem(String upc) {      public Double getTotalReceipts() {      public Double getTotalReceipts(LocalDate start , LocalDate end) {    public Double getTotalPayments() {    public Double getTotalPayments(LocalDate start, LocalDate end) {    public Long setUpSale(Long customerID,String phoneNumber, int cashierID)    private Customer getCustRecord (Long customerID, String phoneNumber)  public Long completeSale(Long salesOrderID,String phoneNumber, int cashierID,ArrayList<SalesOrderDetail> details,Double minPurchase)        public SalesOrderDetail scanItem(String upc, int qty) {      public void addToSale(Long salesOrderID,SalesOrderDetail theDetail)  public void removeFromSales(Long salesOrderID, SalesOrderDetail theDetail)    public void sellItem(String upc, int qty,int cashierID) {        public ArrayList<LowItem> checkRestockLevels(){    public void modifyPhoneNumber (Long ID, String currPhoneNO, String newPhoneNO)            public Long addCustomer (String firstName, String lastName,Address address,String phoneNumber,char gender) {      public String toString()    }  printReceipt(Long salesOrderID)  processReturn(itemID, qty, customerID, salesOrderID)  Methods that can provide answers for data analytics such as  getBestCustomer()  getBestSellingItem()  getVendorWithMostItems() | Instantiate a Store  Store all the contents in a File using object serialization  Add Item to Inventory  Add reference to Vendor to collection of Vendors  Modify the Inventory to restock the Item  Modify the PurchaseOrder to record that the item has been received  Modify the Inventory to restock each Item in the collection of ReceivedItems.  Modify the PurchaseOrder to note that each of these Items has been received  Instantiate a new PurchaseOrder  Add the PurchaseOrder to the Vendor  Add the PurchaseOrder to the collection of Store’s PurchaseOrders  Return the value of the purchaseOrderID  Instantiate a new PurchaseOrder and add to Store’s purchases.  For each ItemOrder, verify that the Item is ordered from the Vendor indicated.  If yes, instantiate a PurchaseOrderDetail for the Item being ordered and add that detail to the PurchaseOrder  Add a PurchaseOrderDetail for this Item to an existing PurchaseOrder with ‘poID’  Instantiate a Payment that is a RECEIVABLE  Examine each PaymentDetail  If the PaymentDetail is made with an EBT ensure that the total paid using EBT doesn’t exceed the amount of EBT eligible Items purchased with the SalesOrder for which this Payment is being made  If the Payment was made by a Customer, modify the Customer balance to reflect this total Payment  Add the Payment to the collection of Store payments.  Modify the SalesOrder to reflect receipt of this Payment  Instantiate a Payment that is a PAYABLE if the VendorID and purchaseOrderID represent valid data.  Instantiate a PaymentDetail to reflect the details of this payment.  Modify the PurchaseOrder to reflect this Payment  Add this payment to the Store’s payments  Modify the restockLevel of the Item with ‘upc’  Return true Store has an Item with ‘upc’ in Inventory  Sum all Payments that are RECEIVABLEs  return total receipts in a certain range of dates  return total Payments that are PAYABLEs  return total payments made within a range of dates to Vendors  instantiate a SalesOrder , if the Customer information is missing that is acceptable  add the SalesOrder to collection of sales  return the salesOrderID  search for the Customer record by his phoneNumber. Verify that the correct Customer has been retrieved by comparing the customerID  find the SalesOrder with salesOrderID  if none exists, throw exception  if order was already completed , throw exception  modify the SalesOrder and add the SalesOrderDetails if any appear in the list  Apply any discounts that are applicable  Mark the order, completed  If a Customer is associated with the SalesOrder apply the SalesOrder invoke Customer method to apply the SalesOrderDetails to the Customer balance  Determine current price of the Item with upc  Determine is a discount price applies to the Item with upc at the current time  Instantiate a SalesOrderDetail identifying all the details about the sale of this Item,including regular price and discounted price, if one applies  Modify the Inventory to reduce inventory level of this item  Add the SalesOrderDetail if the SalesOrder exists and if it hasn’t yet been completed  Remove the SalesOrderDetail if the SaleOrder exists and it hasn’t yet been completed  Designed to handle quick sale of one Item  Instantiate a SalesOrderDetail  set up a sale identifying just the cashier , note the salesOrderID of this new sale.  complete the sale, providing the salesOrderID and SalesOrderDetail that was instantiated  return list of Items whose inventory level has fallen below the restocklevel  find the Customer with the current phone number, modify the Customer’s phone number to reflect the new phone number.  Remove the Customer record mapped to the old phone number  Insert the Customer record so that it is mapped to the new phone number  Ensure that the system doesn’t include a Customer with this phoneNumber  Instantiate a new Customer  Insert the Customer into collection organized by phone number and collection organized by ID  Return the customer id  Print a receipt for the Customer displaying all the items purchased and total paid, including method of payment. Customer info may be omitted if none was provided  Search for the itemID in the SalesOrderDetails of that sales order.  Retrieve price paid and multiply by qty returned.  This may depend if item was on sale and if sale price was applied at the time of Sale.  Adjust the customer balance to apply credit for the item returned |  |

|  |  |  |
| --- | --- | --- |
| Class | Inventory |  |
| Superclass |  |  |
| Subclass |  |  |
| Responsibilities |  | Collaborators |
| Maintain the “state” of the Inventory  Manage the collection of Items that are part of the Inventory | Provide constructor, getters and methods to manage the purchase and sale of Items. | Item  HashMap  LowItem |
| reStockItem(upc, qtyReceived)  sellItem(upc, qty)  addNewItem(Item)  getCurrentPrice(upc)  getDiscountedPrice(upc)  ArrayList <LowItem> getBelowRestockLevel  Boolean hasItem(upc)  Item getItem(String upc)  setRestockLevel (upc, qty)  methods that can be used for data analytics such as getItemWithMostSales() | Add the qtyReceived to the qtyInInventory of the item with that upc.  If Item can’t be found throw an exception  Subtract qty from the qtyInInventory of the item with upc.  If Item can’t be found throw an exception  Add a new Item to the inventory list, items.  Return current price of the Item with upc  Refer to latest discount related to the Item, if one exists. Verify if current date is within the range of dates specified for the discounted price. If there is no discounted price applicable return null.  Return a collection of LowItem , each Item whose current quantity is less than the reorderLevel value  Return true if have Item with that upc, otherwise return false  Return a deep copy of the Item with upc  Modify the restock level of the Item with the upc |  |

A close up of text on a white background

Description automatically generated

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **CRITERIA** | **Exemplary** | **Very Good** | **Acceptable** | **Partially Acceptable** | **Unsatisfactory** | **Raw** | **Weight** | **Total** |
|  | **5 points** | **4 points** | **3 points** | **2 points** | **0-1 points** |  |  |  |
| **Specifications** | Code runs, produces correct results in **all** cases, meets all specifications | Code runs, produces correct results in **all** cases, doesn’t include at most 1 specification | Code runs, produces correct results but doesn’t include at most 2 specifications | Code runs, does not always produce correct results, does not meet more than 2 specifications, “bombs” when run under 1-2 conditions | Code produces incorrect results and “bombs” under more than two conditions |  | **4** |  |
| **Readability** | Code is exceptionally well organized and easy to follow, variables and methods are assigned meaningful names | Code is well organized, each method performs a specific task | Code is fairly easy to read and understand | Code can be understood and followed only by someone who is familiar with the specifications | Code is hard to follow and understand |  | **1** |  |
| **Reusability** | Class code can be reused, application is versatile | Most of the class code or methods can be reused | Fair amount of the class code or methods can be reused | Some part of the code could be reused | Code is not written in a fashion that enables it to be reusable |  | **1** |  |
| **Documentation** | Documentation is clear, concise and explains what code is accomplishing and how  Each method is prefaced with documentation, parameters and return value are documented | Most of the code has been clearly documented  At most two significant methods aren’t prefaced by documentation outlining parameters and return value | Some high level documentation is provided | Minimal amount of documentation is provided and the documentation doesn’t explain the code well | Documentation is trivial and doesn’t serve a practical purpose |  | **2** |  |
| **Delivery** | Code was delivered on time | Code was delivered within one week of due date | Code was delivered one week after due date | Code was delivered within 2 weeks of due date | Code was more than 2 weeks overdue |  | **2** |  |
| **Efficiency** | Code is extremely efficient without sacrificing understanding  Most efficient data structure has been used to store and organize data given the type of access required. | Code is efficient, data structure that is most efficient was not chosen or data structure was not implemented in the most efficient manner | Code is fairly efficient without sacrificing understanding  At least one calculation is repeated unnecessarily | Code is repetitive, unnecessarily long, brute force is used to implement certain features | Code appears to be patched together |  | **1** |  |
| **User Interface** | Professional appearance, easy to use, very intuitive  Results are displayed clearly, in pleasing manner  Student took initiative to develop a GUI | Professional appearance but not intuitive  GUI is not as easy to use or navigate | Easy to use, appearance is not as professional  Results are displayed less clearly  GUI was not developed | Appearance is amateurish and interface isn’t intuitive nor easy to use  Results are displayed less effectively | Interface is not effective  Sufficient menu options aren’t provided  Pertinent results are not displayed at all |  | **3** |  |
| **Testing** | Shows evidence that code was tested exhaustively , includes JUnit test cases and/or UI that uses each feature  exception handling was included to handle all cases | Interface includes options to test all specifications however insufficient evidence that developer tested each option and combinations of options. This might be evidenced by erroneous results that are produced | Interface includes options to test most specifications  Exception handling handles most cases | Interface includes options to test few specifications  Many exception conditions are not handled | Code shows minimal or no evidence that it was tested  Exception conditions were not handled |  | **3** |  |
| **OOD**  **Class/**  **Responsibility/**  **Collaboration** | Effectively implemented OOD by successfully identifying CRC  Differentiated properly between class, instance and local method variables  Each method focused on accomplishing one main task and code reuse was applied in all cases | Utilized OOD, made a minimal amount of erroneous judgements when defining CRC, used class, instance and local variables correctly | Utilized OOD but made some erroneous judgments when defining CRC or used some instance variables in place of local method variables  Some methods duplicated code segments | Utilized OOD but made many erroneous judgments when defining CRC and allowed direct access to instance variables undermining encapsulation and data hiding | OOD principles not implemented |  | **3** |  |