Final Exam
Xiaopei Zhang A20302816
Q1.
Single Responsibility Principle: We made a single DBConnection class so we can initialize it easily every time. This class has one single purpose to connect with Fourier server. The only one reason to change this class is that we change our connection to database.

```
import java.sql.*;

public class DBConnection {

   private Connection conn;
   private String url ="jdbc:oracle:thin:@fourier.cs.iit.edu:1521:ORCL";
   private String user = "pvinay";;
   private String password = "Donotreveal";

   public Connection getDBConnection()  {
      try {
         Class.forName("oracle.jdbc.driver.OracleDriver");
      }
            catch (ClassNotFoundException e) {
               e.printStackTrace();
            }

            try {
               url = "jdbc:oracle:thin:@fourier.cs.iit.edu:1521:ORCL";
               user = "pvinay";
               password = "Donotreveal";
               conn = DriverManager.getConnection(url, user, password);
            }
      catch (SQLException e) {
         e.printStackTrace();
      }
                  return conn;
   }
}
```

Open Closed Principle: setSize() method is polymorphically used by different components with different parameters. All the objects that uses setSize() are subclasses of Component class. They extends Component without modifying it.

1.set size for frame

```
    public OrderHistory(String customerNumber) {
        setTitle("Order History");
        setLayout(null);
        setBounds(300, 100, 300, 300);
    ……
```

2.set size for button

```
        JButton b2 = new JButton("Cancel");
        b2.setSize(90, 30);
    ……
```

3.set size for check box

```
                JCheckBox cb1;
                cb1 = new JCheckBox("" + itemName);
                center.add(cb1);
                cb1.setSize(150, 50);
    ……
```

4.set size for label

```
                JLabel label4 = new JLabel("" + itemId);
                center.add(label4);
                label4.setSize(150, 50);
    ……
```

Liskov Substitution Principle: We wrote AdminFrame for admin to check and modify items, and we needed another highly similar frame CustomerFrame for customer to browse items and shop. Instead of extending AdminFrame and modifying content pane in OrderFrame, OrderFrame is made independently from AdminFrame. The reason that we'd better not use inheritance here even given this high similarity is that AdminFrame should be replaceable with the instance of its subclass without altering the correctness of that program, but obviously CustomerFrame would need many modifications of components and action listeners to replace AdminFrame.
Please note that these two highly similar classes are 200 lines each, so I only put partial code below. To sum up, both frames have exactly the same search function. While AdminFrame has modifying items and order status functions, CustomerFrame has adding items to cart and checking order history functions.

AdminFrame

```
public class AdminFrame extends JFrame {

        private JCheckBox cb1;
        private JCheckBox cb2;
        private JCheckBox cb3;
        private JTextField jt;


        public AdminFrame() {
                RetailItemSearchCriteria criteria = new RetailItemSearchCriteria(true, true, true);
                defalultConstructor(criteria);
        }

        public AdminFrame(RetailItemSearchCriteria criteria) {
                defalultConstructor(criteria);

        }

        public ImageIcon addImage(String url) {
…...
        }

        private void defalultConstructor(RetailItemSearchCriteria criteria) {
                setTitle("Admin Frame");
                setBounds(100, 100, 900, 1600);
                BorderLayout border = new BorderLayout();
                Container content = getContentPane();
                content.setLayout(border);
```

```
    }

…...

                JButton search = new JButton("Search");
                search.addActionListener(new SearchButton());
                JButton add = new JButton("Add one new item");
                add.addActionListener(new AddButton());
                JButton order = new JButton("Order Status Update");
                order.addActionListener(new OrderButton());
                jt = new JTextField("Order number");

…...

                JPanel center = new JPanel(new GridLayout(4, 9));

                ImageIcon buttonIcon;

                DBConnection c = new DBConnection();
                Connection conn;

                try {
…...

                content.add(center, BorderLayout.CENTER);
                setVisible(true);
                setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        }

        private class SearchButton implements ActionListener {
…...
        }

        private class AddButton implements ActionListener {
…...
        }

        private class ItemButton implements ActionListener {
…...
        }

        private class OrderButton implements ActionListener {
…...
        }
}
```

CustomerFrame

```
public class CustomerFrame extends JFrame {

        private JCheckBox cb1;
        private JCheckBox cb2;
        private JCheckBox cb3;
        private String customerID;

        public CustomerFrame(String customerNumber) {
                RetailItemSearchCriteria criteria = new RetailItemSearchCriteria(true, true, true);
                defalultConstructor(customerNumber, criteria);
        }

        public CustomerFrame(String customerNumber, RetailItemSearchCriteria criteria) {
                defalultConstructor(customerNumber, criteria);

        }

        public ImageIcon addImage(String url) {
…...
        }

        private void defalultConstructor(String customerNumber, RetailItemSearchCriteria
criteria) {
                customerID = customerNumber;
                setTitle("Customer Frame");
                setBounds(100, 100, 900, 1600);
                BorderLayout border = new BorderLayout();
                Container content = getContentPane();
                content.setLayout(border);


…...


                JButton search = new JButton("Search");
                search.addActionListener(new SearchButton());
                JButton cart = new JButton("Cart");
                cart.addActionListener(new CartButton());
                JButton order = new JButton("Order History");
                order.addActionListener(new OrderButton());
                JButton login;
                if(customerNumber.equals("0")) {
                        login = new JButton("Login");
```

```
                    login.addActionListener(new LoginButton());
                    north.add(login);
                }

…...

                JPanel center = new JPanel(new GridLayout(4, 9));

                ImageIcon buttonIcon;

                DBConnection c = new DBConnection();
                Connection conn;

                try {
…...

                content.add(center, BorderLayout.CENTER);
                setVisible(true);
                setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
        }

        private class SearchButton implements ActionListener {
…...
        }

        private class CartButton implements ActionListener {
…...
        }

        private class ItemButton implements ActionListener {
…...
        }

        private class OrderButton implements ActionListener {
…...
        }

        private class LoginButton implements ActionListener {
…...
        }
}
```

Dependency Inversion Principle: Because of extending AbstractButton, JButton’s function addActionListener() depends on abstract interface ActionListener, not concrete button handler class, for example, OrderButton, which implements interface ActionListener.

```
        JButton order = new JButton("Order History");
        order.addActionListener(new OrderButton());
        …...
    private class OrderButton implements ActionListener {
        public void actionPerformed(ActionEvent arg0) {
            if(customerID.equals("0")) {
                JOptionPane.showMessageDialog(null, "You must login to
continue.");
                dispose();
                new CustomerPortal();

            }
            else {
                new OrderHistory(customerID);
            }
        }
    }
```

Interface Segregation Principle: Interfaces RootPaneContainer, and ActionListener are separate, which is better than one interface containing all the methods in these two interfaces. getContentPane(), and actionPerformed() methods do not have to be implemented together in one frame. For example, JFrame implements RootPaneContainer so PortalFrame can use getContentPane(), while PortalFrame does not have to implement ActionListener. We can choose to implement ActionListener in either PortalFrame or private button handler classes like pressB1. Below we choose to implement ActionListener in private classes like pressB1.

```
public class PortalFrame extends JFrame {

        public PortalFrame() {

                setTitle("E-Store System");
                setLayout(null);
                setBounds(500, 300, 230, 150);
                Container c = getContentPane();

                JLabel jl = new JLabel("Choose your identity",JLabel.CENTER);
                jl.setHorizontalAlignment(SwingConstants.CENTER);
                jl.setOpaque(true);
                jl.setSize(210, 50);
                c.add(jl);

                JButton b1 = new JButton("Admin");
                b1.setBounds(10, 60, 90, 30);
                b1.addActionListener(new pressB1());
                c.add(b1);

                JButton b2 = new JButton("Customer");
                b2.setBounds(110, 60, 90, 30);
                b2.addActionListener(new pressB2());
                c.add(b2);

                JButton b3 = new JButton("I am just browsing.");
                b3.setBounds(30, 90, 160, 30);
                b3.addActionListener(new pressB3());
                c.add(b3);

                setVisible(true);
                setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        }

        private class pressB1 implements ActionListener {
```

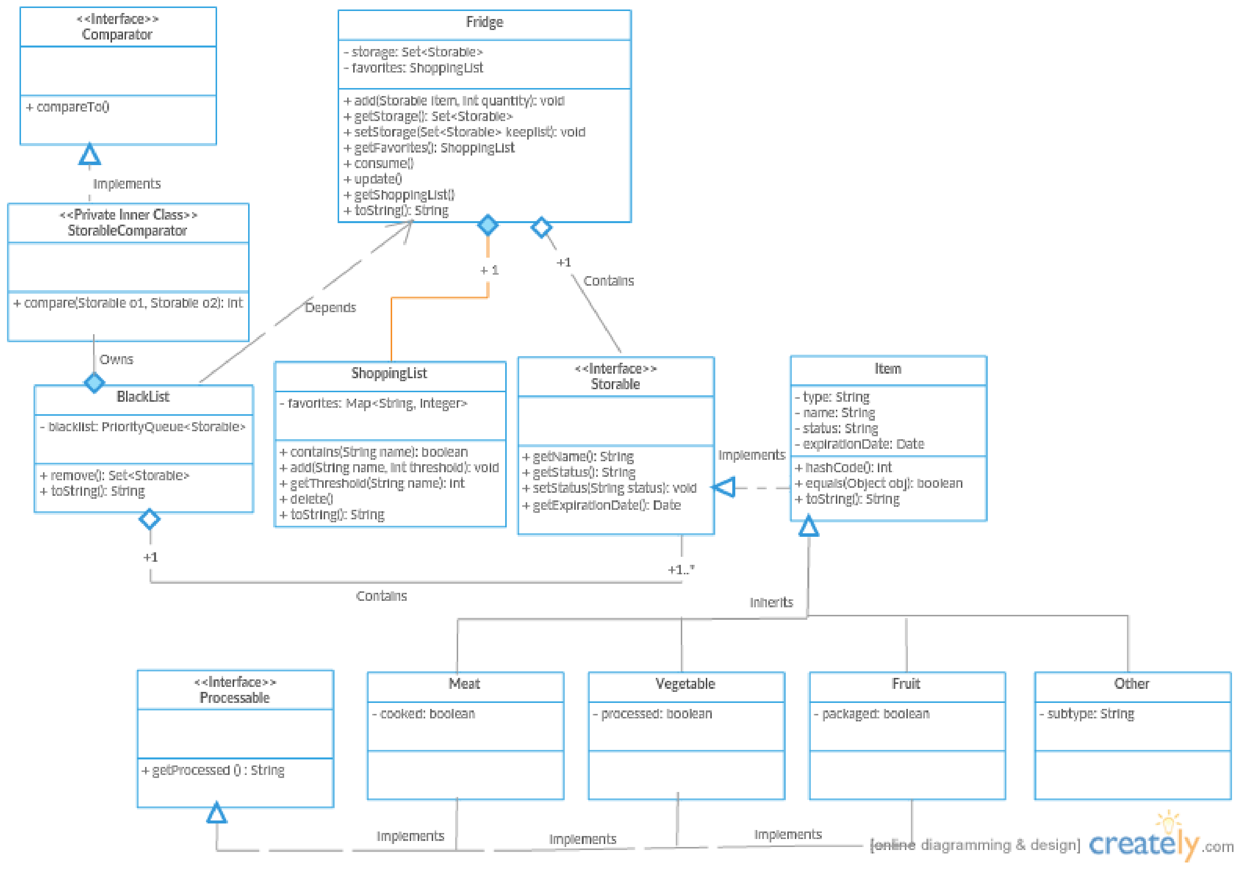
```
		public void actionPerformed(ActionEvent arg0) {
			dispose();
			new AdminPortal();
		}
	}

	private class pressB2 implements ActionListener{
		public void actionPerformed(ActionEvent arg0) {
			dispose();
			new CustomerPortal();
		}
	}

	private class pressB3 implements ActionListener{
		public void actionPerformed(ActionEvent arg0) {
			dispose();
			new CustomerFrame("0");
		}
	}

}
```

Q2.
1.



2. High cohesion is preferred, and it means that fields that are closely related are encapsulated together in one class. For example, in my design the type of each item, e.g., meat, vegetable, fruit or other, and the status of each item, e.g., low, medium and high, are both item information, so they are in the same class Item. I did not and should not separately store these information in some other class.
Low coupling is also preferred, and it means that different classes do not have strong relations. For example, my ShoppingList and BlackList are two separate classes. Although class ShoppingList is owned by class Fridge and class BlackList depends on the set of items in class Fridge, they are not highly and directly related. Therefore, they are two different classes.

3. add(Storable item, int quantity) method is in class Fridge. This method assumes that all the items to be added to storage must exist in or be added to favorites. The field favorites is a ShoppingList which stores the name and the threshold for each item. 1) If the item is already in favorites and storage, the status of this item will change based on the threshold: If it was low, it at least increases to medium, and increases to high if the quantity is more than threshold; if it was medium, increase to high. 2) If the item is in favorites but not in storage, the item's status will be updated according to the quantity: below threshold is low, at threshold is medium, above

threshold is high, and then this item will be added to storage. 3) If the item is not in favorites, since I assume that one item must be added to favorites before storage, so it cannot be in storage either. I will take this quantity as threshold, set the status to medium, and add it to both favorites and storage.
The rest of classes and interfaces are listed for reference. Please note that all the classes are partially implemented, and some functions like getters, setters and those in the UML in question 1 are not implemented for the sole purpose of answering this question concisely.

Class Fridge

```
import java.util.Set;
import java.util.HashSet;

public class Fridge {

    private Set<Storable> storage;
    private ShoppingList favorites;

    public Fridge() {
        storage = new HashSet<>();
        favorites = new ShoppingList();
    }

    public void add(Storable item, int quantity) {
        // if item is in favorites
        if(favorites.contains(item.getName())) {
            // get its threshold
            int threshold = favorites.getThreshold(item.getName());
            // if item already exist in storage
            // modify status
            // assume medium as amount of threshold
            // more than threshold is high
            // adding to low makes at least medium
            if(storage.contains(item)) {
                storage.remove(item);
                if(item.getStatus().equals("medium")) {
                    item.setStatus("high");
                }
                else {
                    if(item.getStatus().equals("low")) {
                        if(quantity > threshold) {
                            item.setStatus("high");
                        }
                        else {
                            item.setStatus("medium");
```

```
                                        }
                                }
                        }
                        storage.add(item);
                }
                // not in storage, add item
                else {
                        if(quantity > threshold) {
                                storage.add(item);
                        }
                        else if(quantity == threshold) {
                                storage.add(item);
                        }
                        else {
                                item.setStatus("low");
                        }
                        storage.add(item);
                }
        }
        // not in favorites, cannot be in storage either
        // add item to both
        else {
                // assume threshold to be current quantity
                favorites.add(item.getName(), quantity);
                storage.add(item);
        }
}

public Set<Storable> getStorage() {
        return this.storage;
}

public void setStorage(Set<Storable> keeplist) {
        this.storage = (Set<Storable>)keeplist;
}

public ShoppingList getFavorites() {
        return this.favorites;
}

public String toString() {
        String result = "";
        for(Storable item: storage) {
                result += item.toString() + "\n";
            }
            return result;
        }

}
```

Class ShoppingList

```
import java.util.HashMap;

public class ShoppingList {

        private HashMap<String, Integer> favorites;

        public ShoppingList() {
            favorites = new HashMap<>();
        }

        public boolean contains(String name) {
            return favorites.containsKey(name);
        }

        public void add(String name, int threshold) {
            favorites.put(name, threshold);
        }

        public int getThreshold(String name) {
            return favorites.get(name);
        }

        public String toString() {
            String result = "";
            for(String s: favorites.keySet()) {
                result += s + " " + String.valueOf(favorites.get(s)) + "\n";
            }
            return result;
        }

}
```

Interface Storable

```
import java.util.Date;
```

```
public interface Storable {

        public String getName();
        public String getStatus();
        public void setStatus(String status);
        public Date getExpirationDate();

}
```

Class Item

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Item implements Storable{

        private String type;     // meat, vegetable, fruit, other
        private String name;
        private String status;            // low, medium, high
        private Date expirationDate;

        public Item(String type, String name, String expiration) throws ParseException {
                this.type = type;
                this.name = name;
                this.status = "medium";
                this.expirationDate = new SimpleDateFormat("yyyy-MM-dd").parse(expiration);
        }

        @Override
        public String getName() {
                return name;
        }

        @Override
        public String getStatus() {
                return this.status;
        }

        @Override
        public void setStatus(String newStatus) {
                this.status = newStatus;
        }
```

```
	@Override
	public Date getExpirationDate() {
		return expirationDate;
	}

	@Override
	public int hashCode() {
		int result = this.type.hashCode();
		result *= 11;
		result += this.name.hashCode();
		result *= 11;
		result += this.expirationDate.hashCode();
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if(getClass() != obj.getClass()) {
			return false;
		}
		Item item = (Item) obj;
		return this.type.equals(item.type) && this.name.equals(item.name) &&
this.expirationDate.equals(item.expirationDate);
	}

	public String toString() {
		return type + " " + name + " " + status + " " + new
SimpleDateFormat("yyyy-MM-dd").format(expirationDate);
	}

}
```

Class Meat

```
import java.text.ParseException;

public class Meat extends Item implements Processable {

	boolean cooked;

	public Meat(String name, String expiration, String cooked) throws ParseException {
		super("meat", name, expiration);
		this.cooked = Boolean.parseBoolean(cooked);
	}
	@Override
	public String getProcessed() {
		if(cooked) {
			return "cooked";
		}
		else {
			return "raw";
		}
	}

	@Override
	public int hashCode() {
		int result = super.hashCode();
		result *= 11;
		result += (this.cooked? 0 : 1);
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if(getClass() != obj.getClass()) {
			return false;
		}
		Meat meat = (Meat) obj;
		return super.equals(meat) && this.cooked == meat.cooked;
	}

	public String toString() {
		return super.toString() + " " + getProcessed();
	}

}
```

Class Vegetable

```
import java.text.ParseException;

public class Vegetable extends Item implements Processable {

	boolean processed;

	public Vegetable(String name, String expiration, String processed) throws
ParseException {
		super("vegetable", name, expiration);
		this.processed = Boolean.parseBoolean(processed);
	}

	@Override
	public String getProcessed() {
		if(processed) {
			return "processed";
		}
		else {
			return "fresh";
		}
	}

	@Override
	public int hashCode() {
		int result = super.hashCode();
		result *= 11;
		result += (this.processed? 0 : 1);
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if(getClass() != obj.getClass()) {
			return false;
		}
		Vegetable vege = (Vegetable) obj;
		return super.equals(vege) && this.processed == vege.processed;
	}

	public String toString() {
		return super.toString() + " " + getProcessed();
	}

}
```

Class Fruit

```
import java.text.ParseException;

public class Fruit extends Item implements Processable {

	boolean packaged;
	public Fruit(String name, String expiration, String packaged) throws ParseException {
		super("fruit", name, expiration);
		this.packaged = Boolean.parseBoolean(packaged);
	}

	@Override
	public String getProcessed() {
		if(packaged) {
			return "packaged";
		}
		else {
			return "fresh";
		}
	}

	@Override
	public int hashCode() {
		int result = super.hashCode();
		result *= 11;
		result += (this.packaged? 0 : 1);
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if(getClass() != obj.getClass()) {
			return false;
		}
		Fruit fruit = (Fruit) obj;
		return super.equals(fruit) && this.packaged == fruit.packaged;
	}

	public String toString() {
		return super.toString() + " " + getProcessed();
	}

}
```

Class Other

```
import java.text.ParseException;

public class Other extends Item {
	String subtype;

	public Other(String name, String expiration, String subtype) throws ParseException {
		super("other", name, expiration);
		this.subtype = subtype;
	}

	@Override
	public int hashCode() {
		int result = super.hashCode();
		result *= 11;
		result += subtype.hashCode();
		return result;
	}

	@Override
	public boolean equals(Object obj) {
		if(getClass() != obj.getClass()) {
			return false;
		}
		Other other = (Other) obj;
		return super.equals(other) && this.subtype.equals(other.subtype);
	}

	public String toString() {
		return super.toString() + " " + subtype;
	}

}
```

Interface Processable

```
public interface Processable {

	public String getProcessed();

}
```

4. The hierarchy of classes in this design would be that four classes Meat, Vegetable, Fruit and Other all inherit class Item. Each class with its own attributes is listed below. All the four classes have all the attributes Item class has.
Item
--

- type: String
- name: String
- status: String
- expirationDate: Date

Meat
--
- cooked: boolean

Vegetable
--
- processed: boolean

Fruit
--
- packaged: boolean

Other
--
- subtype: String

5. An aggregation relationship exists between class Fridge and class Item which implements Storable. Class Fridge stores lots of storable items entered by the user, while they both conceptually independently exist. No one has to exist for the other to exist.

6. My class BlackList takes in a set of storable items, and uses its method remove() to return a set of items that should be keeped and leave all the expired items in the PriorityQueue blacklist.
Class BlackList

```
import java.util.Set;
import java.util.Comparator;
import java.util.Date;
import java.util.HashSet;
import java.util.Iterator;
import java.util.PriorityQueue;

public class BlackList {

        private PriorityQueue<Storable> blacklist;

        public BlackList(Set<Storable> storage) {
                blacklist = new PriorityQueue<>(new StorableComparator());
                for(Storable item: storage) {
                        blacklist.add(item);
```

```
		}
	}

	public Set<Storable> remove() {
		Set<Storable> keeplist = new HashSet<>();
		while(!blacklist.isEmpty() && blacklist.peek().getExpirationDate().compareTo(new
Date()) > 0) {
			keeplist.add(blacklist.poll());
		}
		return keeplist;
	}

	public String toString() {
		String result = "";
		Iterator<Storable> itr = blacklist.iterator();
		while(itr.hasNext()) {
			result += itr.next().toString() + "\n";
		}
		return result;
	}

	private class StorableComparator implements Comparator<Storable> {

		@Override
		public int compare(Storable o1, Storable o2) {
			return o2.getExpirationDate().compareTo(o1.getExpirationDate());
		}

	}

}
```

7. As shown in the code above in question 6, I implemented a priority queue with a comparator StorableComparator to handle the reordering of storable items. Class StorableComparator compares two Storable objects by their expiration date. In the priority queue, items would be listed from the latest expiration date to the earliest expiration date. This queue will always first pop the item that expires later than the rest of the items in the queue. In the method remove(), if the top item of the queue expired when compared with the current date, all the rest of the items in the queue must be expired as well. Therefore, remove() could use the priority queue to pop out all the unexpired items and leave all the expired items in the queue, and return a set of items that can be kept.