2021 Software Engineering Design Examination Solution

1a

Below is the content in a new file called Order.java. It is an abstract class that serves as the template for both SmallOrder and BulkOrder

```
// Order.java
package retail;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.List;

public abstract class Order {
    protected final List<Product> items;
    protected final CreditCardDetails creditCardDetails;
    protected final Address billingAddress;
    protected final Address shippingAddress;
    protected final Courier courier;

    public Order(List<Product> items, CreditCardDetails creditCardDetails, Address
billingAddress, Address shippingAddress, Courier courier) {
        this.items = items;
        this.creditCardDetails = creditCardDetails;
        this.billingAddress = billingAddress;
        this.shippingAddress = shippingAddress;
        this.courier = courier;
    }

    public abstract void process();

    protected BigDecimal calculateItemPrice() {
        BigDecimal total = new BigDecimal(0);

        for (Product item : items) {
            total = total.add(item.unitPrice());
        }

        return total;
    }

    protected void charge(BigDecimal amount, CreditCardDetails creditCardDetails,
Address billingAddress) {
        CreditCardProcessor.getInstance().charge(round(amount), creditCardDetails,
billingAddress);
```

```
    }

    protected BigDecimal round(BigDecimal amount) {
        return amount.setScale(2, RoundingMode.CEILING);
    }
}
```

We can then change the implementation of SmallOrder and BulkOrder as follows

```
// SmallOrder.java
package retail;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Collections;
import java.util.List;

public class SmallOrder extends Order {

  private static final BigDecimal GIFT_WRAP_CHARGE = new BigDecimal(3);

  private final boolean giftWrap;

  public SmallOrder(
      List<Product> items,
      CreditCardDetails creditCardDetails,
      Address billingAddress,
      Address shippingAddress,
      Courier courier,
      boolean giftWrap) {
    super(items, creditCardDetails, billingAddress, shippingAddress, courier);
    this.giftWrap = giftWrap;
  }

  @Override
  public void process() {

    BigDecimal total = super.calculateItemPrice();

    total = total.add(courier.deliveryCharge());

    if (giftWrap) {
      total = total.add(GIFT_WRAP_CHARGE);
    }

    charge(round(total), creditCardDetails, billingAddress);

    courier.send(giftWrap ? new GiftBox(items) : new Parcel(items), shippingAddress);
```

```
  }
}
```

```
// BulkOrder.java
package retail;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Collections;
import java.util.List;

public class BulkOrder extends Order {

  private final BigDecimal discount;

  public BulkOrder(
      List<Product> items,
      CreditCardDetails creditCardDetails,
      Address billingAddress,
      Address shippingAddress,
      Courier courier,
      BigDecimal discount) {
    super(items, creditCardDetails, billingAddress, shippingAddress, courier);
    this.discount = discount;
  }

  @Override
  public void process() {

    BigDecimal total = super.calculateItemPrice();

    if (items.size() > 10) {
      total = total.multiply(BigDecimal.valueOf(0.8));
    } else if (items.size() > 5) {
      total = total.multiply(BigDecimal.valueOf(0.9));
    }

    total = total.subtract(discount);

    charge(round(total), creditCardDetails, billingAddress);

    courier.send(new Parcel(items), shippingAddress);
  }
}
```

1b

We have created an OrderBuilder class to help improve the structure of code.

```
// OrderBuilder.java (a new class)
package retail;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

public class OrderBuilder {

    enum OrderType {
        BULK, SMALL
    }

    private List<Product> items = new ArrayList<>();
    private CreditCardDetails creditCardDetails;
    private Address billingAddress;
    private Address shippingAddress;
    private Courier courier;
    private BigDecimal discount = new BigDecimal(0);
    private boolean giftWrap = false;

    // default set to SMALL
    private OrderType type = OrderType.SMALL;

    private OrderBuilder(List<Product> items, CreditCardDetails creditCardDetails,
                         Address billingAddress, Courier courier) {
        // These fields are required on each order
        this.items = items;
        this.creditCardDetails = creditCardDetails;
        this.billingAddress = billingAddress;

        // shipping address is the same as billing address when not provided
        this.shippingAddress = billingAddress;
        this.courier = courier;

        // set the type of order to BULK if it's more than 3
        if (items.size() > 3) {
            type = OrderType.BULK;
        }
    }

    public static OrderBuilder anOrder(List<Product> items, CreditCardDetails
creditCardDetails,
                                       Address billingAddress, Courier courier) {
        return new OrderBuilder(items, creditCardDetails, billingAddress, courier);
```

```
    }

    public Order build() {
        if (type.equals(OrderType.BULK)) {
            return new BulkOrder(items, creditCardDetails, billingAddress,
shippingAddress, courier, discount);
        } else {
            return new SmallOrder(items, creditCardDetails, billingAddress,
shippingAddress, courier, giftWrap);
        }
    }

    // Able to add one item at a time. This function needs more refactoring as we need
to be more careful to deal with the situation of conversion between SMALL and BULK
orders.
    public OrderBuilder withItem(Product product) {
        this.items.add(product);
        if (items.size() > 3) {
            this.type = OrderType.BULK;
        }
        return this;
    }

    public OrderBuilder withShippingAddress(Address address) {
        this.shippingAddress = address;
        return this;
    }

    public OrderBuilder withGiftWrap() throws Exception {
        if (!type.equals(OrderType.SMALL)) {
            throw new Exception("Cannot add gift wrap to bulk orders!");
        }
        this.giftWrap = true;
        return this;
    }

    public OrderBuilder withDiscount(BigDecimal discount) throws Exception {
        if (!type.equals(OrderType.BULK)) {
            throw new Exception("Cannot add discount to small orders!");
        }
        this.discount = discount;
        return this;
    }
}
```

1c

i) Singleton Object

ii) Singleton Object can cause tight coupling between different classes. In this case, the class Order is tightly coupled with the class CreditCardProcessor because Order mentions the classname of CreditCardProcessor. In this way, it is hard to swap the charging process to another implementation or interface. Additionally, it is harder to write unit test as well, as we cannot mock a CreditCardProcessor object and pass it in the class Order.

iii) To apply the dependency inversion principle, we create an abstraction of credit card payment system, namely a class called `PaymentSystem`. Below is it's implementation.

```
// PaymentSystem.java
package retail;

import java.math.BigDecimal;

public interface PaymentSystem {
    public void charge(BigDecimal amount, CreditCardDetails details, Address
billingAddress);
}
```

Then we add a paymentSystem field in Order.java

```
// Order.java
...
// This is added in order to dismiss the tight coupling between this class and
CreditCardPRocessor
    protected PaymentSystem paymentSystem;

    public Order(List<Product> items, CreditCardDetails creditCardDetails, Address
billingAddress, Address shippingAddress, Courier courier, PaymentSystem paymentSystem)
{
        this.items = items;
        this.creditCardDetails = creditCardDetails;
        this.billingAddress = billingAddress;
        this.shippingAddress = shippingAddress;
        this.courier = courier;
        this.paymentSystem = paymentSystem;
    }
...
```

Then we write a test to verify if the credit card is charged properly

```
// RetailTest.java
package retail;
```

```
import org.jmock.Expectations;
import org.jmock.integration.junit4.JUnitRuleMockery;
import org.junit.Rule;
import org.junit.Test;

import java.math.BigDecimal;
import java.util.List;

public class RetailTest {

    @Rule
    public final JUnitRuleMockery context = new JUnitRuleMockery();

    CreditCardProcessor processor = context.mock(CreditCardProcessor.class);
    Courier courier = context.mock(Courier.class);

    @Test
    public void smallOrderChargedCorrectly() throws Exception {
        Product product =
                new Product("One Book", new BigDecimal("10.00"));
        CreditCardDetails creditCard = new CreditCardDetails("1234123412341234", 111);
        Address addr = new Address("180 Queens Gate, London, SW7 2AZ");
        Order order = OrderBuilder.anOrder(List.of(product), creditCard, addr, courier,
processor).withGiftWrap().build();

        context.checking(new Expectations(){{
            oneOf(processor).charge(new BigDecimal(13), creditCard, addr);
        }});

        order.process();
    }

    @Test
    public void bigOrderChargedCorrectly() {
        List<Product> list = List.of(
                new Product("One Book", new BigDecimal("10.00")),
                new Product("One Book", new BigDecimal("10.00")),
                new Product("One Book", new BigDecimal("10.00")),
                new Product("One Book", new BigDecimal("10.00")),
                new Product("One Book", new BigDecimal("10.00")),
                new Product("One Book", new BigDecimal("10.00")));
        CreditCardDetails creditCard = new CreditCardDetails("1234123412341234", 111);
        Address addr = new Address("180 Queens Gate, London, SW7 2AZ");

        Order order = OrderBuilder.anOrder(list, creditCard, addr, courier,
processor).build();

        context.checking(new Expectations(){{
            oneOf(processor).charge(new BigDecimal(54), creditCard, addr);
```

```
        }});

        order.process();
    }
}
```

2a

We create a class called TennisModel as shown below

```
package tennis;

public class TennisModel {
    private int playerOneScore;
    private int playerTwoScore;

    private final String[] scoreNames;

    public TennisModel(String[] scoreNames) {
        this.playerOneScore = 0;
        this.playerTwoScore = 0;
        this.scoreNames = scoreNames;
    }

    public String score() {

        if (playerOneScore > 2 && playerTwoScore > 2) {
            int difference = playerOneScore - playerTwoScore;
            switch (difference) {
                case 0:
                    return "Deuce";
                case 1:
                    return "Advantage Player 1";
                case -1:
                    return "Advantage Player 2";
                case 2:
                    return "Game Player 1";
                case -2:
                    return "Game Player 2";
            }
        }

        if (playerOneScore > 3) {
            return "Game Player 1";
        }
        if (playerTwoScore > 3) {
            return "Game Player 2";
        }
        if (playerOneScore == playerTwoScore) {
```

```
            return scoreNames[playerOneScore] + " all";
        }
        return scoreNames[playerOneScore] + " - " + scoreNames[playerTwoScore];
    }

    public void playerOneWinsPoint() {
        playerOneScore++;
    }

    public void playerTwoWinsPoint() {
        playerTwoScore++;
    }

    public boolean gameHasEnded() {
        return score().contains("Game");
    }
}
```

2b

As an example, those are three tests we can add

```
package tennis;

import org.junit.Test;

public class TennisTest {
    @Test
    public void playerOneWillAdvantage() {
        TennisModel model = new TennisModel(new String[]{"player1", "player2"});

        model.playerOneWinsPoint();
        model.playerOneWinsPoint();
        model.playerOneWinsPoint();
        model.playerOneWinsPoint();

        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();

        assertEquals(model.score(), "Advantage Player 1");
    }

    @Test
    public void playerTwoWillGame() {
        TennisModel model = new TennisModel(new String[]{"player1", "player2"});

        model.playerOneWinsPoint();
        model.playerOneWinsPoint();
```

```
        model.playerOneWinsPoint();

        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();

        assertEquals(model.score(), "Game Player 2");
    }

    @Test
    public void playerOneWillGame() {
        TennisModel model = new TennisModel(new String[]{"player1", "player2"});

        model.playerOneWinsPoint();
        model.playerOneWinsPoint();
        model.playerOneWinsPoint();
        model.playerOneWinsPoint();

        model.playerTwoWinsPoint();
        model.playerTwoWinsPoint();

        assertEquals(model.score(), "Game Player 1");
    }
}
```

2c

We create a class called `TennisView.java` as shown below

```
package tennis;

import javax.swing.*;
import java.awt.event.ActionListener;

public class TennisView {

    public final JButton playerOneScores = new JButton("Player One Scores");
    public final JButton playerTwoScores = new JButton("Player Two Scores");
    public final JTextField scoreDisplay = new JTextField(20);

    public TennisView(ActionListener player1, ActionListener player2) {
        JFrame window = new JFrame("Tennis");
        window.setSize(400, 150);

        scoreDisplay.setHorizontalAlignment(JTextField.CENTER);
        scoreDisplay.setEditable(false);
```

```
        playerOneScores.addActionListener(player1);
        playerTwoScores.addActionListener(player2);

        JPanel panel = new JPanel();
        panel.add(playerOneScores);
        panel.add(playerTwoScores);
        panel.add(scoreDisplay);

        window.add(panel);

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

    }

    public void update(String textDisplay) {
      scoreDisplay.setText(textDisplay);
    }
}
```

Then we change the implementation in `TennisScorer.java`:

```
package tennis;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TennisScorer {
    private TennisView view = new TennisView(new PlayerOneController(), new
PlayerTwoController());
    private TennisModel model = new TennisModel(new String[]{"player 1", "player 2"});

    class PlayerOneController implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            model.playerOneWinsPoint();
            view.scoreDisplay.setText(model.score());
            if (model.gameHasEnded()) {
                view.playerOneScores.setEnabled(false);
                view.playerTwoScores.setEnabled(false);
            }
        }
    }

    class PlayerTwoController implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            model.playerTwoWinsPoint();
            view.scoreDisplay.setText(model.score());
            if (model.gameHasEnded()) {
                view.playerOneScores.setEnabled(false);
```

```
                view.playerTwoScores.setEnabled(false);
            }
        }
    }

    public static void main(String[] args) {
        // leave blank as we can run the program upon initialization of this class
    }
}
```

In the `TennisView.java` class, we add a field called `List<TennisView> observers` and add corresponding method for adding new views.

2d

As an example, the test should be something looks like this (still need a better answer here :))

```
...
    @Rule
    JUnitRuleMockery context = new JUnitRuleMockery();
    TennisView view = context.mock(TennisView.class);

    @Test
    public void viewIsUpdated() {
        TennisModel model = new TennisModel(new String[]{});
        model.addObserver(view);
        context.checking(new Expectations(){{
            oneOf(view).update("Something");
        }});

        new TennisScorer();
    }
...
```