//////////////////////////////////////////////////////////////

// A.

//////////////////////////////////////////////////////////////

i)

Reusability: By separating concerns and dependencies,

it is much easier to reuse for later applications.

E.g. the model could be ported and integrated into a different graphical interface.

ii)

Separation of Concerns: by separating the code into 3 different elements, it is much easier for different people to work on it at the same time.

///////////////////////////////////////////////

// ABSTRACT BASE CLASS

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 = Collections.unmodifiableList(items);

this.creditCardDetails = creditCardDetails;

this.billingAddress = billingAddress;

this.shippingAddress = shippingAddress;

this.courier = courier;

}

public abstract void process();

protected BigDecimal round(BigDecimal amount) {

return amount.setScale(2, RoundingMode.CEILING);

}

}

//////////////////////////////////////////////////

// SMALLORDER

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 = new BigDecimal(0);

for (Product item : items) {

total = total.add(item.unitPrice());

}

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

if (giftWrap) {

total = total.add(GIFT\_WRAP\_CHARGE);

}

CreditCardProcessor.getInstance().charge(round(total), creditCardDetails, billingAddress);

if (giftWrap) {

courier.send(new GiftBox(items), shippingAddress);

} else {

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

}

}

}

//////////////////////////////////////////////////

// BULKORDER

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 = new BigDecimal(0);

for (Product item : items) {

total = total.add(item.unitPrice());

}

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);

CreditCardProcessor.getInstance().charge(round(total), creditCardDetails, billingAddress);

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

}

}

//////////////////////////////////////////////////////////////

// B

//////////////////////////////////////////////////////////////

public class OrderBuilder {

private List<Product> items = null;

private CreditCardDetails creditCardDetails = null;

private Address billingAddress = null;

private Address shippingAddress = null;

private Courier courier = null;

private BigDecimal discount = null;

private boolean giftWrap = false;

private OrderBuilder() {}

public static OrderBuilder config() { return new OrderBuilder();}

public Order build() throws IllegalArgumentException {

if (items == null) {

System.out.println("There are no items to be shipped. Order canceled.");

throw new IllegalArgumentException();

}

if (billingAddress == null) {

System.out.println("Address required. Order canceled.");

throw new IllegalArgumentException();

}

if (courier == null) {

System.out.println("No courier specified. Order canceled.");

throw new IllegalArgumentException();

}

if (creditCardDetails == null) {

System.out.println("No card details provided. Order canceled.");

throw new IllegalArgumentException();

}

if (shippingAddress == null) shippingAddress = billingAddress;

if (items.size() <= 3) {

return new SmallOrder(

items,

creditCardDetails,

billingAddress,

shippingAddress,

courier,

giftWrap);

}

else {

if (discount == null) discount = BigDecimal.ZERO;

return new BulkOrder(

items,

creditCardDetails,

billingAddress,

shippingAddress,

courier,

discount);

}

}

public OrderBuilder setItem(Product item) {

items.add(item);

return this;

}

public OrderBuilder setCreditCardDetails(CreditCardDetails details) {

creditCardDetails = details;

return this;

}

public OrderBuilder setBillingAddress(Address address) {

billingAddress = address;

return this;

}

public OrderBuilder setShippingAddress(Address address) {

shippingAddress = address;

return this;

}

public OrderBuilder setCourier(Courier courier) {

this.courier = courier;

return this;

}

public OrderBuilder setDiscount(BigDecimal discount) {

this.discount = discount;

return this;

}

public OrderBuilder addGiftWrap() {

giftWrap = true;

return this;

}

}

//////////////////////////////////////////////////////////

/\*

Orderbuilder Demo with same orders as provided code

\*/

public static void main(String[] args) {

Courier royalMail = new RoyalMail();

Courier fedex = new Fedex();

Order bigOrder = OrderBuilder.config()

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setCreditCardDetails(new CreditCardDetails("1234123412341234", 111))

.setBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))

.setCourier(fedex)

.build();

bigOrder.process();

Order smallOrder = OrderBuilder.config()

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setCreditCardDetails(new CreditCardDetails("1234123412341234", 111))

.setBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))

.setCourier(royalMail)

.build();

smallOrder.process();

}

//////////////////////////////////////////////////////////////

// C

//////////////////////////////////////////////////////////////

i)

Singleton Pattern.

ii)

it creates a global variable. global vars are not desireable

because they are hard to reason about, introduce dependencies

across the object graph and make it difficult to reuse the code.

Also they are hard to test.

iii)

// created a Processor interface

public interface Processor {

void charge(BigDecimal price, CreditCardDetails creditCardDetails, Address billingAddress);

}

/\*

In the base class and interface, process() method is updated with

a parameter that takes a Processor interface object.

\*/

public abstract void process(Processor processor);

//in the Order subclasses, process is implemented

//e.g. SmallOrder implementation. Singleton is replaced with interface.

// BulkOrder not copied as same change.

@Override

public void process(Processor processor) {

//common part of the method is refactored into a method in base class

//to avoid duplication, see below for definition

BigDecimal total = calcBasePrice();

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

if (giftWrap) {

total = total.add(GIFT\_WRAP\_CHARGE);

}

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

if (giftWrap) {

courier.send(new GiftBox(items), shippingAddress);

} else {

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

}

}

//Furthermore, to avoid changing client code, the CreditCardProcessor class is wrapped

//into an adapter class that implements the Processor interface

public class CreditCardProcessorAdapter implements Processor{

@Override

public void charge(BigDecimal price, CreditCardDetails account,

Address address) {

CreditCardProcessor.getInstance().charge(price, account, address);

}

}

//calcBasePrice definition in base class Order

public abstract class Order{

protected BigDecimal calcBasePrice() {

BigDecimal total = new BigDecimal(0);

for (Product item : items) {

total = total.add(item.unitPrice());

}

return total;

}

}

//Unit Test for Order class. Basically took the same orders as in the

//RetailExample.

public class OrderTest {

Courier royalMail = new RoyalMail();

Courier fedex = new Fedex();

@Rule

public JUnitRuleMockery context = new JUnitRuleMockery();

private final Processor processor = context.mock(Processor.class);

@Test

public void moreThanThreeItemsAreProcessedAsBulkOrder() {

Order bigOrder = OrderBuilder.config()

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setCreditCardDetails(new CreditCardDetails("1234123412341234", 111))

.setBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))

.setCourier(fedex)

.build();

assertThat(bigOrder, instanceOf(BulkOrder.class));

}

@Test

public void lessOrEqualThanThreeItemsAreProcessedAsSmallOrder() {

Order smallOrder = OrderBuilder.config()

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setCreditCardDetails(new CreditCardDetails("1234123412341234", 111))

.setBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))

.setCourier(royalMail)

.build();

assertThat(smallOrder, instanceOf(SmallOrder.class));

}

@Test

public void processorIsChargedWhenOrderGetsProcessed() {

context.checking(new Expectations() {{

exactly(1).of(processor).charge(

with(any(BigDecimal.class)),

with(any(CreditCardDetails.class)),

with(any(Address.class)));

}});

Order order = OrderBuilder.config()

.setItem(new Product("One Book", new BigDecimal("10.00")))

.setCreditCardDetails(new CreditCardDetails("1234123412341234", 111))

.setBillingAddress(new Address("180 Queens Gate, London, SW7 2AZ"))

.setCourier(royalMail)

.build();

order.process(processor);

}

}

//////////////////////////////////////////////////////////////////

// A

//////////////////////////////////////////////////////////////////

public class TennisScorer {

Model m = new Model();

public static void main(String[] args) {

new TennisScorer().display();

}

private void display() {

JFrame window = new JFrame("Tennis");

window.setSize(400, 150);

JButton playerOneScores = new JButton("Player One Scores");

JButton playerTwoScores = new JButton("Player Two Scores");

JTextField scoreDisplay = new JTextField(20);

scoreDisplay.setHorizontalAlignment(JTextField.CENTER);

scoreDisplay.setEditable(false);

playerOneScores.addActionListener(

e -> {

m.playerOneWinsPoint();

scoreDisplay.setText(m.score());

if (m.gameHasEnded()) {

playerOneScores.setEnabled(false);

playerTwoScores.setEnabled(false);

}

});

playerTwoScores.addActionListener(

e -> {

m.playerTwoWinsPoint();

scoreDisplay.setText(m.score());

if (m.gameHasEnded()) {

playerOneScores.setEnabled(false);

playerTwoScores.setEnabled(false);

}

});

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);

}

}

//-----------------------------------------------------------------

// Pulled out Model class

public class Model {

private int playerOneScore = 0;

private int playerTwoScore = 0;

private final String[] scoreNames = {"Love", "15", "30", "40"};

public Model() {}

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");}

}

//////////////////////////////////////////////////////////////////

// B

//////////////////////////////////////////////////////////////////

public class ModelTest {

@Test

public void playerOneWinsPointUpdatesScore() {

Model m = new Model();

m.playerOneWinsPoint();

assertEquals("15 - Love", m.score());

}

@Test

public void playerTwoWinsPointUpdatesScore() {

Model m = new Model();

m.playerTwoWinsPoint();

assertEquals("Love - 15", m.score());

}

@Test

public void modelRecognizesWhenPlayerHasWon() {

Model m = new Model();

m.playerOneWinsPoint();

m.playerOneWinsPoint();

m.playerOneWinsPoint();

m.playerOneWinsPoint();

assertEquals(true, m.gameHasEnded());

}

}

//////////////////////////////////////////////////////////////////

// C

//////////////////////////////////////////////////////////////////

//First, we need two interfaces to set up observer pattern.

public interface Observer {

void update(Model model);

}

public interface Observable {

void notifyObservers();

}

//--------------------------------------------------------------

//The Model is updated to implement the Observable interface

class Model implements Observable{

public void addObserver(Observer o) {

observers.add(o);

}

@Override

public void notifyObservers() {

for(Observer o : observers) {

o.update(this);

}

}

//these two methods are updated to notify observer

//when smt changes.

public void playerOneWinsPoint() {

playerOneScore++;

notifyObservers();

}

public void playerTwoWinsPoint() {

playerTwoScore++;

notifyObservers();

}

}

//--------------------------------------------------------------

//A new View class is created to handle all the GUI stuff, implementing

//the Observer interface.

public class View implements Observer{

private final JButton playerOneScores;

private final JButton playerTwoScores;

JTextField scoreDisplay;

public View() {

JFrame window = new JFrame("Tennis");

window.setSize(400, 150);

playerOneScores = new JButton("Player One Scores");

playerTwoScores = new JButton("Player Two Scores");

scoreDisplay = new JTextField(20);

scoreDisplay.setHorizontalAlignment(JTextField.CENTER);

scoreDisplay.setEditable(false);

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 addControllerB1(ActionListener controller) {

playerOneScores.addActionListener(controller);

}

public void addControllerB2(ActionListener controller) {

playerTwoScores.addActionListener(controller);

}

@Override

public void update(Model m) {

scoreDisplay.setText(m.score());

if (m.gameHasEnded()) {

playerOneScores.setEnabled(false);

playerTwoScores.setEnabled(false);

}

}

}

//--------------------------------------------------------------

//This allows the TennisScorer to function as the controller, whose

//only responsibility is to wire the view to the model and instantiate

//the app.

public class TennisScorer {

Model m = new Model();

View v = new View();

private TennisScorer() {

m.addObserver(v);

v.addControllerB1(e -> {

m.playerOneWinsPoint();

});

v.addControllerB2(e -> {

m.playerTwoWinsPoint();

});

}

public static void main(String[] args) {

new TennisScorer();

}

}

//////////////////////////////////////////////////////////////////

// D

//////////////////////////////////////////////////////////////////

//In ModelTest class, add following test:

@Rule

public JUnitRuleMockery context = new JUnitRuleMockery();

private final Observer o = context.mock(Observer.class);

@Test

public void pressingButtonWillUpdateObserver() {

Model m = new Model();

context.checking(new Expectations() {{

exactly(1).of(o).update(m);

}});

m.addObserver(o);

m.playerOneWinsPoint();

}