**Q1**

a) Use of a Singleton Pattern

b)

**MarkingSystem Class**

public class MarkingSystem {  
  
 // SINGLETON PATTERN: Add static attribute instance as object of same Class  
 private static MarkingSystem *instance*;   
 private final Map<Student, StudentRecord> allStudentMarks;  
   
 // SINGLETON PATTERN: Make Constructor of class private  
 private MarkingSystem() {  
 allStudentMarks = new HashMap<>();  
 }  
   
 // SINGLETON PATTERN: create method getInstance() considering lazy initialization  
 public static synchronized MarkingSystem getInstance(){ // synchronized used for doing lazy initialization  
 if(*instance* == null) // (one thread can call this method, not two)  
 *instance* = new MarkingSystem(); // (method is being locked for 1 thread)  
 return *instance*;  
 }

…

**ExamResult Class**

**…**

public void scaleAndSave(double scalingFactor) {  
 MarkingSystem.*getInstance*().registerMark(this, scalingFactor);  
}

**...**

c) This pattern has two initialization variants:

* **Eager Initialization**: singleton instance is initialized at beginning of program when class is loaded. To get access to the instance a getInstance() method is used.

public static MarkingSystem getInstance(){  
 return *instance*;  
 }

* **Lazy Initialization**: singleton instance is initialized when it is needed (by calling method getInstance(), that creates instance if it does not exist). For this case we have to be careful, only allowing one thread to call this method (syncronized keyword used)

public static synchronized MarkingSystem getInstance(){  
 if(*instance* == null)  
 *instance* = new MarkingSystem();  
 return *instance*;  
}

d) The use of a singleton produces many problems:

* Introduce a global variable -> In general this kind of variables are not welcome because they are hard to reason about
* Introduce dependencies into our system that span widely along the object graph
* Introduce thigh coupling (dependency)
  + Make difficult to reuse code
  + Make testing hard -> testing in isolation is not possible because singleton is necessary

**Q2**

a) Use of a Template Pattern using inheritance. This introduces tight coupling.

b) An alternative would be the strategy pattern, which allows loose coupling by introducing composition.

c) Change Class CodeFormatter from abstract to not abstract, create Interface Language and change logic of classes JavaCodeFormatter and RubyCodeFomatter to JavaLanguage and RubyLanguage. Modify also classes JavaCodeFormatterTest and RubyCodeFormatterTest for instantiation of javaFormatter and rubyFormatter variables.

**CodeFormatter Class**

package Q2;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class CodeFormatter {  
  
 private Language language;  
  
 public CodeFormatter(Language language){  
 this.language = language;  
 }  
  
  
 public String format(String source) {  
  
 String trimmed = stripBlankLines(source);  
 int indentLevel = 0;  
  
 List<String> indentedCode = new ArrayList<>();  
  
 for(String line : linesOf(trimmed)) {  
 if (line.contains(language.endOfBlock())) {  
 indentLevel -= 1;  
 }  
 indentedCode.add(indentBy(indentLevel, language.tabsOrSpaces(), line));  
 for (String openBlock : language.startOfBlock()) {  
 if (line.contains(openBlock)) {  
 System.*out*.println("line contains " + openBlock);  
 indentLevel += 1;  
 }  
 }  
 }  
  
 return String.*join*("\n", indentedCode);  
 }  
  
 private String indentBy(int num, WhiteSpace whiteSpace, String line) {  
 String indent = "";  
 for(int i = 0; i < num; i++) {  
 indent = indent + whiteSpace.literal;  
 }  
 return indent + line.trim();  
 }  
  
 private List<String> linesOf(String source) {  
 return Arrays.*asList*(source.split("\n"));  
 }  
  
 private String stripBlankLines(String source) {  
 return source.trim();  
 }  
  
}

**Language Interface**

package Q2;  
  
import java.util.List;  
  
public interface Language {  
  
 List<String> startOfBlock();  
 String endOfBlock();  
 WhiteSpace tabsOrSpaces();  
}

**JavaLanguage Class**

package Q2;  
  
import java.util.List;  
  
public class JavaLanguage implements Language {  
  
 @Override  
 public List<String> startOfBlock() {  
 return List.*of*("{");  
 }  
  
 @Override  
 public String endOfBlock() {  
 return "}";  
 }  
  
 @Override  
 public WhiteSpace tabsOrSpaces() {  
 return WhiteSpace.*TWOSPACES*;  
 }  
  
}

**RubyLanguage Class**

package Q2;  
  
import java.util.List;  
  
  
public class RubyLanguage implements Language {  
  
 @Override  
 public List<String> startOfBlock() {  
 return List.*of*("do", "if", "while");  
 }  
  
 @Override  
 public String endOfBlock() {  
 return "end";  
 }  
  
 @Override  
 public WhiteSpace tabsOrSpaces() {  
 return WhiteSpace.*TABS*;  
 }  
}

**JavaCodeFormatterTest Class**

**…**

CodeFormatter javaFormatter = new CodeFormatter(new JavaLanguage());

**…**

**RubyCodeFormatterTest Class**

**…**

CodeFormatter rubyFormatter = new CodeFormatter(new RubyLanguage());

**…**

d) The strategy pattern is much more preferred than the template pattern.

* The latter one introduces tight coupling between the parent and child classes, due to the use of inheritance.
* Although duplication is removed using both patterns, when using the latter one, it is not possible to have separated independent reusable components.
* The strategy pattern avoids this problem by using composition, introducing more flexibility due to the looser coupling created between components.

**Q3**

a) IOmplement class BoxOffice and Interfaces Payments, Theatre and WaitingList. Also implement BoxOfficeTest class. i), ii) and iii) below:

**BoxOffice Class**

package Q3;  
  
public class BoxOffice {  
  
 private final Theatre theatre;  
 private final Payments payments;  
 private final WaitingList waitingList;  
  
 public BoxOffice(Theatre theatre, Payments payments, WaitingList waitingList){  
 this.theatre = theatre;  
 this.payments = payments;  
 this.waitingList = waitingList;  
  
 }  
  
 public void bookTickets(Show show, int numberOfTickets, Customer customer){  
 if(theatre.checkAvailability(show,numberOfTickets)){  
 payments.pay(show.price()\*numberOfTickets, customer);  
 theatre.confirm(show,numberOfTickets);  
 }  
 }  
  
 public void returnTickets(Show show, int numberOfTickets){  
 waitingList.anyoneWaiting(show, numberOfTickets);  
 }  
  
 public void bookTickets(Customer customer, Show show, int numberOfTickets){  
 payments.pay(show.price()\*numberOfTickets, customer);  
 }  
}

**Payments Interface**

package Q3;  
  
public interface Payments {  
 void pay(double cost, Customer customer);  
}

**Theatre Interface**

package Q3;  
  
public interface Theatre {  
 boolean checkAvailability(Show show, int numberOfTickets);  
 void confirm(Show show, int numberOfTickets);  
}

**WaitingList Interface**

package Q3;  
  
public interface WaitingList {  
 boolean anyoneWaiting(Show show, int numberOfTickets);  
}

**BoxOfficeTest Class**

package Q3;  
  
import org.jmock.Expectations;  
import org.jmock.integration.junit4.JUnitRuleMockery;  
import org.junit.Rule;  
import org.junit.Test;  
  
public class BoxOfficeTest {  
  
 static final Show *LION\_KING* =  
 new Show("The Lion King", 44.00);  
  
 static final Show *HAMILTON* =  
 new Show("Hamilton", 80.00);  
  
 static final Customer *SALLY* = new Customer("Sally Davies");  
 static final Customer *TOM* = new Customer("Thomas Williams");  
  
 @Rule  
 public JUnitRuleMockery context = new JUnitRuleMockery();  
 Payments payments = context.mock(Payments.class);  
 Theatre theatre = context.mock(Theatre.class);  
 WaitingList waitingList = context.mock(WaitingList.class);  
  
  
 @Test  
 public void bookFourTicketsForLionKing(){  
  
 BoxOffice boxOffice = new BoxOffice(theatre, payments, waitingList);  
 context.checking(new Expectations(){{  
 exactly(1).of(theatre).checkAvailability(*LION\_KING*,4);  
 will(*returnValue*(true));  
 exactly(1).of(payments).pay(*LION\_KING*.price()\*4,*SALLY*);  
 exactly(1).of(theatre).confirm(*LION\_KING*,4);  
 }});  
  
 boxOffice.bookTickets(*LION\_KING*,4,*SALLY*);  
 }  
  
 @Test  
 public void bookTwoTicketsForHamilton(){  
  
 BoxOffice boxOffice = new BoxOffice(theatre, payments, waitingList);  
 context.checking(new Expectations(){{  
 exactly(1).of(theatre).checkAvailability(*HAMILTON*,2);  
 will(*returnValue*(false));  
 }});  
  
 boxOffice.bookTickets(*HAMILTON*,2,*TOM*);  
 }  
  
 @Test  
 public void returnTickets(){  
  
 BoxOffice boxOffice = new BoxOffice(theatre, payments, waitingList);  
 context.checking(new Expectations(){{  
 exactly(1).of(waitingList).anyoneWaiting(*HAMILTON*,4);  
 exactly(1).of(payments).pay(*HAMILTON*.price()\*2,*TOM*);  
 }});  
  
 boxOffice.returnTickets(*HAMILTON*,4);  
 boxOffice.bookTickets(*TOM*,*HAMILTON*,2);  
 }  
  
}

b)

i) Both are different types of messages that allow communication between objects in a system.

* A query is a request to another object in which we expect a return value, so then we can use it.
  + These messages should not have any kind of side effects on the state of the object invoked.
* A command, on the other hand, is an instruction that one object sends to another object for delegating a task.
  + Normally no return values are sent back.
  + It often changes the state of the invoked object or another part of the program.

ii) A query for example will be the method *checkAvailability(show, quantity)* that sends a query from a *BoxOffice* to a *Theatre* object, expecting the return of a boolen value. A command, on the other hand, would be the method *confirm(show, quantity)* between the same objects. In this case no return value is expected. Just a change in the state of the *Theatre* Object.

**Q4**

**a)**

i) Use of Adapter Pattern

ii) Create class *WordlPayDotComPaymentService* and Interface *PaymentService*. Then modify class *ShoppingBasket* so all instance and methods from third-party library are taken inside adapter, and then only adapter interacts with *ShoppingBasket* class

**WorldPayDotComPaymentService Class**

package ic.doc;  
  
 import com.worldpay.CardNumber;  
import com.worldpay.CreditCardTransaction;  
import com.worldpay.TransactionProcessor;  
  
// ADAPTER PATTERN: Implementing Adapter Class  
public class WorldPayDotComPaymentService implements PaymentService {  
 private CardNumber cardNumber;  
  
 // Methods to override  
 @Override  
 public void enterCardNumber(String cardNumberDetails) {  
 this.cardNumber = new CardNumber(cardNumberDetails);  
 }  
  
 @Override  
 public void processTransaction(int pounds, int pence){  
 CreditCardTransaction transaction = new CreditCardTransaction(cardNumber, pounds, pence);  
 new TransactionProcessor().process(transaction);  
 }  
  
}

**PaymentService Interface**

package ic.doc;  
  
import com.worldpay.CardNumber;  
import com.worldpay.CreditCardTransaction;  
import com.worldpay.TransactionProcessor;  
  
// ADAPTER PATTERN: Implementing Interface  
public interface PaymentService {  
  
 // Methods to implement  
 void enterCardNumber(String cardNumberDetails);  
 public void processTransaction(int pounds, int pence);  
}

**ShoppingBasket Class**

package ic.doc;  
  
import com.worldpay.CardNumber;  
import com.worldpay.CreditCardTransaction;  
import com.worldpay.TransactionProcessor;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class ShoppingBasket {  
  
 private final Map<Item, Integer> items = new HashMap<>();  
 private PaymentService paymentService; // ADAPTER PATTERN: Taking out CardNumber class attribute  
 // Replacing it with Adapter, and putting that attribute  
 // inside adapter  
  
  
 public void addItem(Item item) {  
 if (items.containsKey(item)) {  
 items.put(item, items.get(item) + 1);  
 } else {  
 items.put(item, 1);  
 }  
 }  
  
 public void enterCardDetails(String cardNumber) {  
 // ADAPTER PATTERN: inserting adapter for initialization of CardNumber attribute  
 paymentService.enterCardNumber(cardNumber);  
 }  
  
 public void checkout() {  
 int totalPounds = 0;  
 int totalItems = 0;  
 for (Item item : items.keySet()) {  
 Integer quantity = items.get(item);  
 totalItems = totalItems + quantity;  
 totalPounds = totalPounds + quantity \* item.priceInPounds();  
 }  
  
 if (totalItems > 3) {  
 totalPounds = Math.*min*(totalPounds, totalPounds - 5);  
 }  
  
 // ADAPTER PATTERN: We get rid of instances of third-party library objects that run methods  
 // from that library. Instead we delegate task using adapter to interact with those instances  
 // and methods doing same work, but without polluting our code.  
 paymentService.processTransaction(totalPounds, 0);  
 }  
}

iii) Architectural Style used here is the Hexagonal Architecture

**b)** Answer in a) ii)

**c)** Create Class *WorldPayDotComPaymentServiceTest* and *ShoppingBasketTest* with some tests inside

**WorldPayDotComPaymentService Class**

package ic.doc;  
  
import org.junit.Test;  
  
import static junit.framework.TestCase.*fail*;  
import static org.hamcrest.CoreMatchers.*containsString*;  
import static org.hamcrest.MatcherAssert.*assertThat*;  
  
public class WorldPayDotComPaymentServiceTest {  
  
 // Test correct integration of Adapter with Third-Party Library  
  
 @Test  
 public void creditCardDetailsAreNotCorrectlyEntered(){  
  
 PaymentService service = new WorldPayDotComPaymentService();  
  
 try {  
 service.enterCardNumber("");  
 *fail*("should have thrown error");  
 } catch (IllegalArgumentException ex){  
 *assertThat*(ex.getMessage(), *containsString*("card number must be 16 digits"));  
 }  
 }  
  
 @Test  
 public void creditCardDetailsAreCorrectlyEntered(){  
  
 PaymentService service = new WorldPayDotComPaymentService();  
 try {  
 service.enterCardNumber("1234567891234567");  
 } catch (IllegalArgumentException ex){  
 *assertThat*(ex.getMessage(), *containsString*("card number must be 16 digits"));  
 }  
 }  
   
}

**ShoppingBasketTest Class**

package ic.doc;  
  
import org.jmock.Expectations;  
import org.jmock.integration.junit4.JUnitRuleMockery;  
import org.junit.Rule;  
import org.junit.Test;  
  
import java.util.HashMap;  
import java.util.Map;  
  
import static junit.framework.TestCase.*fail*;  
import static org.hamcrest.CoreMatchers.*containsString*;  
import static org.hamcrest.MatcherAssert.*assertThat*;  
  
public class ShoppingBasketTest {  
  
 private static final String *CREDIT\_CARD\_NUMBER* = "1234567891234567";  
 private static final Map<String, Integer> *BASKET* = new HashMap<>(){{  
 put("Teady Bear",5);  
 put("Toy",5);  
 }};  
  
  
 @Rule  
 public JUnitRuleMockery context = new JUnitRuleMockery();  
 PaymentService service = context.mock(PaymentService.class);  
  
 @Test  
 public void creditCardDetailsAreReceivedByPaymentService(){  
 ShoppingBasket shoppingBasket = new ShoppingBasket(service);  
 context.checking(new Expectations(){{  
 exactly(1).of(service).enterCardNumber(*CREDIT\_CARD\_NUMBER*);  
 }});  
 shoppingBasket.enterCardDetails(*CREDIT\_CARD\_NUMBER*);  
 }  
  
 @Test public void canProcessTransaction(){  
 ShoppingBasket shoppingBasket = new ShoppingBasket(service);  
 context.checking(new Expectations(){{  
 exactly(1).of(service).processTransaction(15,0);  
 }});  
 shoppingBasket.addItem(new Item("Teddy Bear",10));  
 shoppingBasket.addItem(new Item("Toy",5));  
 shoppingBasket.checkout();  
 }  
  
}