2020 Software Engineer Design Examination Solution

1. a. i) The first test is shown below

```
public class OrderProcessorTest {
  @Rule
  public JUnitRuleMockery context = new JUnitRuleMockery();
  public Warehouse warehouse = context.mock(Warehouse.class);

  @Test
  public void willCheckStockLevel() {
    OrderProcessor processor = new OrderProcessor(warehouse);
    context.checking(new Expectations(){{
      oneOf(warehouse).checkStockLevel(LEGACY_CODE_BOOK);
    }});

    processor.order(LEGACY_CODE_BOOK, 1, BOB);
  }
}
```

The implementation is then

```
// OrderProcessor.java
package ic.doc;

public class OrderProcessor {
  private Warehouse warehouse;

  public OrderProcessor(Warehouse warehouse) {
    this.warehouse = warehouse;
  }

  public void order(Book book, int count, Customer customer) {
    warehouse.checkStockLevel(book);
  }
}
```

```
// Warehouse.java
package ic.doc;

public interface Warehouse {
  public int checkStockLevel(Book book);
}
```

ii) The second test is shown below

```
public PaymentSystem paymentSystem = context.mock(PaymentSystem.class);
@Test
public void willChargeAndDispatch() {
    OrderProcessor processor = new OrderProcessor(warehouse, paymentSystem);
    context.checking(new Expectations(){{
      oneOf(warehouse).checkStockLevel(DESIGN_PATTERNS_BOOK);
      will(returnValue(3));
      exactly(1).of(paymentSystem).charge(DESIGN_PATTERNS_BOOK.price() * 2, ALICE);
      exactly(1).of(warehouse).dispatch(DESIGN_PATTERNS_BOOK, 2, ALICE);
    }});

    processor.order(DESIGN_PATTERNS_BOOK, 2, ALICE);
}
```

The implementation is then

```
// PaymentSystem.java
package ic.doc;

public interface PaymentSystem {
  public void charge(double amount, Customer customer);
}
```

```
// OrderProcessor.java
package ic.doc;

public class OrderProcessor {
  private Warehouse warehouse;
  private PaymentSystem payment;

  public OrderProcessor(Warehouse warehouse) {
    this.warehouse = warehouse;
  }

  public OrderProcessor(Warehouse warehouse, PaymentSystem payment) {
    this(warehouse);
    this.payment = payment;
  }

  public void order(Book book, int count, Customer customer) {
    warehouse.checkStockLevel(book);
    payment.charge(book.price() * count, customer);
    warehouse.dispatch(book, count, customer);
  }
}
```

```
// Warehouse.java
package ic.doc;

public interface Warehouse {
  public int checkStockLevel(Book book);
  public void dispatch(Book book, int count, Customer customer);
}
```

iii) The third test is shown below

```
@Test
public void willButMoreIfNoneLeft() {
    OrderProcessor processor = new OrderProcessor(warehouse, paymentSystem, buyer);
    context.checking(new Expectations(){{
      oneOf(warehouse).checkStockLevel(LEGACY_CODE_BOOK);
      will(returnValue(0));
      exactly(1).of(buyer).buyMoreOf(LEGACY_CODE_BOOK);

      oneOf(warehouse).checkStockLevel(LEGACY_CODE_BOOK);
      will(returnValue(0));

      oneOf(warehouse).checkStockLevel(LEGACY_CODE_BOOK);
      will(returnValue(5));

      exactly(1).of(paymentSystem).charge(LEGACY_CODE_BOOK.price(), BOB);
      exactly(1).of(warehouse).dispatch(LEGACY_CODE_BOOK, 1, BOB);
    }});

    processor.order(LEGACY_CODE_BOOK, 1, BOB);
    processor.newBookArrived();
    processor.newBookArrived();
}
```

The implementation is then

```
// OrderProcessor.java
package ic.doc;

import java.util.ArrayList;
import java.util.List;

public class OrderProcessor {

  class Order {
    Book book;
    Customer customer;
    int amount;
```

```
    public Order(Book book, Customer customer, int amount) {
      this.book = book;
      this.customer = customer;
      this.amount = amount;
    }
  }

  private Warehouse warehouse;
  private PaymentSystem payment;
  private Buyer buyer;
  private Order pendingOrder;

  public OrderProcessor(Warehouse warehouse) {
    this.warehouse = warehouse;
  }

  public OrderProcessor(Warehouse warehouse, PaymentSystem payment) {
    this(warehouse);
    this.payment = payment;
  }

  public OrderProcessor(Warehouse warehouse, PaymentSystem payment, Buyer buyer) {
    this(warehouse, payment);
    this.buyer = buyer;
  }

  public void order(Book book, int count, Customer customer) {
    if (warehouse.checkStockLevel(book) == 0){
      buyer.buyMoreOf(book);
      pendingOrder = new Order(book, customer, count);
    } else {
      payment.charge(book.price() * count, customer);
      warehouse.dispatch(book, count, customer);
    }
  }

  public void newBookArrived() {
    if (warehouse.checkStockLevel(pendingOrder.book) > 0) {
      payment.charge(pendingOrder.book.price() * pendingOrder.amount,
pendingOrder.customer);
      warehouse.dispatch(pendingOrder.book, pendingOrder.amount,
pendingOrder.customer);
    }
  }
}
```

b. i) Command would tell another object/class to do their jobs without returning any values, creating less coupling as we trust the other class will eventually execute the command. Query would ask another object/class about the information/state of the program so that the other object/class will return a value indicating the state. This will create a tight coupling between classes.

ii) One command would be `processor.newBookArrived()`. One query would be `warehouse.checkStockLevel(Book book)`.

iii) The use of command will result in less coupling between the classes, rather than a chain of queries made across different classes (the train-wreck). Using query freequently will make codes more fragile. For instance, if one component in the chain of query changes, it will highly likely cause breaks in the codebase and need more effort to refactor.

iv) This style of deign is generally called **tell-don't-ask**

2. a. Singleton object. The corresponding code is `StockPrice price = StockMarketDataFeed.getInstance().currentPriceFor(stock)`

   b. i) Adapter pattern. The refactoring looks like this

```
// AlgoTrader.java
package ic.doc;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class AlgoTrader {

  private final List<String> stocksToWatch =
      List.of("GOOG", "MSFT", "APPL");

  private final Map<String, Integer> lastPrices = new HashMap<>();
  private final SimpleBroker broker = new SimpleBroker();
  private final LondonStockExchange exchange = new LondonStockExchange();

  public void trade() {

    for (String stock : stocksToWatch) {

      int price = exchange.currentPriceFor(stock);

      if (isRising(stock, price)) {
        broker.buy(String.valueOf(stock));
      }

      if (isFalling(stock, price)) {
        broker.sell(String.valueOf(stock));
```

```
      }

      lastPrices.put(stock, price);
    }
  }

  private boolean isFalling(String stock, int price) {
    int lastPrice = lastPrices.containsKey(stock) ? lastPrices.get(stock) : 0;
    return price < lastPrice;
  }

  private boolean isRising(String stock, int price) {
    int lastPrice = lastPrices.containsKey(stock) ? lastPrices.get(stock) :
Integer.MAX_VALUE;
    return price > lastPrice;
  }

  public static void main(String[] args) {
    new AlgoTrader().start();
  }

  // code below here is not important for the exam

  private void logPrices(String stock, int price, int lastPrice) {
    System.out.println(
        String.format("%s used to be %s, now %s ", stock, lastPrice, price));
  }

  private void start() {

    // run trade() every minute

    ScheduledExecutorService executorService = Executors.newScheduledThreadPool(1);
    executorService.scheduleAtFixedRate(this::trade, 0, 60, TimeUnit.SECONDS);
  }
}
```

```
// LondonStockExchange.java, this is the adapter class
package ic.doc;

import com.londonstockexchange.StockMarketDataFeed;
import com.londonstockexchange.TickerSymbol;
import java.util.HashMap;
import java.util.Map;

public class LondonStockExchange implements StockExchange {

  private StockMarketDataFeed feed = StockMarketDataFeed.getInstance();
  private Map<String, TickerSymbol> map = new HashMap<>();
```

```
  @Override
  public int currentPriceFor(String stock) {
    TickerSymbol ticker = map.get(stock);
    if (ticker == null) {
      // throw exception
    }
    return feed.currentPriceFor(ticker).inPennies();
  }
}
```

```
// StockExchange.java, this is the adapter interface
package ic.doc;

public interface StockExchange {
  public int currentPriceFor(String stock);
}
```

ii) Hexagonal architecture

c. By the principle of Dependency Inversion (numbered as 1 and 2 below), both `SimpleBroker` and `LondonStockExchange` need to be abstracted (1️⃣ details should depend on abstraction) and passed in from outside (2️⃣ high level module should not depend on low level components), rather than created inside the `AlgoTrader` class. The list of stock to watch needs also to be created non-statically. The first measure will help to create a template(abstraction) that any kind of broker or stock feed can be based upon. The second measure can modularize the code, making it possible to pass other kinds of broker/data feed to the `AlgoTrader` class. Below is the code after changes

```
// AlgoTrader.java
public class AlgoTrader {

  private final List<String> stocksToWatch;

  private final Map<String, Integer> lastPrices;
  private final Broker broker;
  private final StockExchange exchange;

  public AlgoTrader(Broker broker, StockExchange exchange) {
    this.broker = broker;
    this.exchange = exchange;
    lastPrices = new HashMap<>();
  }

  public addToWatchList(String stock) {
    stocksToWatch.add(stock);
  }

  ...
```

```
  public static void main(String[] args) {
    Broker broker = new SimpleBroker();
    StockExchange exchange = new LondonStockExchange();
    AlgoTrader trader = new AlgoTrader(broker, exchange);

    trader.addToWatchList("GOOG", "MSFT", "APPL");

    new AlgoTrader().start();
  }

  ...
}
```

```
// Broker.java, this is an interface
package ic.doc;

public interface Broker {
  public void buy(String stock);
  public void sell(String stock);
}

// where the SimpleBroker implements it
public class SimpleBroker implements Broker {
  ...
}
```

d. Two possible unit tests are

```
JUnitRuleMockery context = new JUnitRuleMockery();
Broker broker = context.mock(SimpleBroker.class);
StockExchange exchange = context.mock(LondonStockExchange.class);

@Test
public void willSellStockOncePriceLow() {
  AlgoTrader trader = new AlgoTrader(broker, exchange);
  context.checking(new Expectations(){{
    oneOf(exchange).currentPriceFor("GOOG");
    will(returnValue(50));
    oneOf(exchange).currentPriceFor("GOOG");
    will(returnValue(20));
    oneOf(broker).sell("GOOG");
  }});

  trader.addToWatchList("GOOG");
  trader.trade();
  trader.trade();
}
```

```
@Test
public void willBuyStockOncePriceLow() {
  AlgoTrader trader = new AlgoTrader(broker, exchange);
  context.checking(new Expectations(){{
    oneOf(exchange).currentPriceFor("GOOG");
    will(returnValue(20));
    oneOf(exchange).currentPriceFor("GOOG");
    will(returnValue(50));
    oneOf(broker).buy("GOOG");
  }});

  trader.addToWatchList("GOOG");
  trader.trade();
  trader.trade();
}
```