1ai)

Test:

@Rule public JUnitRuleMockery context = new JunitRuleMockery();

Warehouse warehouse = context.mock(Warehouse.class);

OrderProcessor orderSystem = new OrderProcessor(warehouse);

b

@Test

public void aNewOrderWillCheckStockLevels() {

context.checking(new Expectations() {{

oneOf(warehouse).checkStockLevel(LEGACY\_CODE\_BOOK);

ignoring(warehouse);

// for later questions, add ignoring(buyer); ignoring(paymentSystem);

}});

orderSystem.order(LEGACY\_CODE\_BOOK, 1, BOB);

}

(OrderProcessor.java)

package ic.doc;

public class OrderProcessor {

private Warehouse warehouse;

public OrderProcessor(Warehouse warehouse) {

this.warehouse = warehouse;

}

void order(Book book, int quantity, Customer customer) {

warehouse.checkStockLevel(book);

}

}

(Warehouse.java)

package ic.doc;

public interface Warehouse {

int checkStockLevel(Book book);

}

1aii)

// New feature: Will just get the OrderProcessor to dispatch and charge after an order is made and stock check is done

Test:

@Rule public JUnitRuleMockery context = new JUnitRuleMockery();

Warehouse warehouse = context.mock(Warehouse.class)

PaymentSystem paymentSystem = context.mock(PaymentSystem.class);

OrderProcessor orderSystem = new OrderProcessor(warehouse, paymentSystem);

@Test

public void bookInStockWillImmediatelyChargeAndDispatch() {

context.checking(new Expectations() {{

oneOf(warehouse).checkStockLevel(DESIGN\_PATTERNS\_BOOK);

will(returnValue(3));

oneOf(warehouse).dispatch(DESIGN\_PATTERNS\_BOOK, 2, ALICE);

oneOf(paymentSystem).charge(DESIGN\_PATTERNS\_BOOK.price() \* 2, ALICE);

}});

orderSystem.order(DESIGN\_PATTERNS\_BOOK, 2, ALICE)

Implementation:

(OrderProcessor.java)

package ic.doc;

public class OrderProcessor {

private final PaymentSystem payment System;

private Warehouse warehouse;

public OrderProcessor(Warehouse warehouse, PaymentSystem paymentSystem) {

this.warehouse = warehouse;

this.paymentSystem = paymentSystem;

}

void order(Book book, int quantity, Customer customer) {

warehouse.checkStockLevel(book);

warehouse.dispatch(book, quantity, customer);

paymentSystem.charge(quantity \* book.price(), customer);

}

}

(PaymentSystem.java)

package ic.doc;

public interface PaymentSystem {

void charge(double price, Customer customer);

}

(Warehouse.java)

package ic.doc;

public interface Warehouse {

int checkStockLevel(Book book);

void dispatch(Book book, int quantity, Customer customer);

}

1aiii) // Changes: Includes no-stock check and introduces Buyer

Test:

@Test

public void bookNotInStockWillAwaitNewDeliveriesTilDispatched() {

context.checking(new Expectations() {{

oneOf(warehouse).checkStockLevel(LEGACY\_CODE\_BOOK);

will(returnValue(0));

oneOf(buyer).buyMoreOf(LEGACY\_CODE\_BOOK);

// buyer now calls newBooksArrived;

oneOf(warehouse).checkStockLevel(LEGACY\_CODE\_BOOK);

will(returnValue(0));

//buyer calls newBooksArrived again;

oneOf(warehouse).checkStockLevel(LEGACY\_CODE\_BOOK);

will(returnValue(5));

oneOf(warehouse).dispatch(LEGACY\_CODE\_BOOK, 1, BOB);

oneOf(paymentSystem).charge(LEGACY\_CODE\_BOOK.price() \* 1, BOB);

}});

orderSystem.order(LEGACY\_CODE\_BOOK, 1, BOB);

orderSystem.newBooksArrived();

orderSystem.newBooksArrived();

}

(Buyer.java)

package ic.doc;

public interface Buyer {

void buyMoreOf(Book book);

}

(OrderProcessor.java)

package ic.doc;

public class OrderProcessor {

private final PaymentSystem paymentSystem;

private final Buyer buyer;

private Warehouse warehouse;

// Note: following principle of simplest implementation!

private Book pendingBook;

private int pendingQuantity;

private Customer pendingCustomer;

public OrderProcessor(Warehouse warehouse, Buyer buyer, PaymentSystem paymentSystem) {

this.warehouse = warehouse;

this.buyer = buyer;

this.paymentSystem = paymentSystem;

}

void order(Book book, int quantity, Customer customer) {

if (warehouse.checkStockLevel(book) == 0) {

buyer.buyMoreOf(book);

this.pendingBook = book;

this.pendingQuantity = quantity;

this.pendingCustomer = customer;

return;

}

warehouse.dispatch(book, quantity, customer);

paymentSystem.charge(quantity \* book.price(), customer);

}

public void newBooksArrived() {

if (warehouse.checkStockLevel(pendingBook) == 0) {

return;

}

warehouse.dispatch(pendingBook, pendingQuantity, pendingCustomer);

paymentSystem.charge(pendingQuantity \* pendingBook.price(), pendingCustomer);

}

}

1bi) When a class 'queries' another class, it delegates the work to another class but it expects a final result back which the caller can use to know that the other class has done its job. When a class 'commands' another class, this delegation is more trust-based in which the caller doesn't expect a response back (can be identified as a void-returning procedure), trusting that the callee class is doing its job as routine.

1bii)

Command: paymentSystem.charge(cost, customer)

Query: warehouse.checkStockLevel(book)

1biii) The usage of commands rather than not using commands (e.g successive chaining across classes and interfaces aka 'trainwrecks') reduces the amount of coupling in the code. Doing 'trainwrecks' will cause classes closer to a client to have to know more and more about the internals of the system and should an internal implementation deep in the codebase require changing, this would very likely break some classes that use the code and may require modifying the existing code in the classes that feature these references.

1biv) Tell, Don't Ask

2a)

i) Singleton

ii) StockMarketDataFeed.getInstance().currentPriceFor(stock);

AlgoTrader relies on the singleton from the Third Party, which creates a tight coupling between the two classes. It becomes very difficult to test the class as there is no enabling point/seam at which we can choose to use the singleton or not.

2bi) Adapter

Implementation:

(AlgoTrader.java)

// Remove all references to 3rd party com.londonstockexchange, using adapter-interface StockExchange

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 StockExchange datafeed = new LondonStockExchangeAdapter();

public void trade() {

for (String stock : stocksToWatch) {

int price = datafeed.currentPriceFor(stock);

if (isRising(stock, price)) {

broker.buy(String.valueOf(stock));

}exams

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);

}

}

(StockExchange.java)

// Interface used by adapters

package ic.doc;

public interface StockExchange {

int currentPriceFor(String stockSymbol);

}

(LondonStockExchangeAdapter.java)

// Handles all communications with 3rd party com.londonstockexchange. Utilises enummap for changing Strings to 3rd party enum

package ic.doc;

import com.londonstockexchange.StockMarketDataFeed;

import com.londonstockexchange.TickerSymbol;

import java.util.EnumMap;

import java.util.Map;

public class LondonStockExchangeAdapter implements StockExchange {

private StockMarketDataFeed stockExchange = StockMarketDataFeed.getInstance();

Map<TickerSymbol, String> map = new EnumMap<>(TickerSymbol.class);

public int currentPriceFor(String symbol) {

if (!map.containsValue(symbol)) {

// throw exception

return 0;

}

for (TickerSymbol ticker: map.keySet()) {

if (ticker.name().equals(symbol)) {

return stockExchange.currentPriceFor(ticker).inPennies();

}

}

return 0;

// throw exception

}

}

ii) Hexagonal Architecture

2ci)

// 1. Added a new interface Broker and made SimpleBroker implement it

// 2. Moved instantiation of Broker and StockExchange out into Main

// 3. Because I believe that a broker and stock exchange data feed is necessary for an AlgoTrader to even work, put them into constructor of AlgoTrader

// 4. Added method to add stocks to watch

(AlgoTrader.java)

private final Broker broker;

private final StockExchange datafeed;

private final List<String> stocksToWatch = new ArrayList<>();

public AlgoTrader(Broker broker, StockExchange datafeed) {

this.broker = broker;

this.datafeed = datafeed;

}

...

public void watchStock(String symbol) {

stocksToWatch.add(symbol);

}

...

public static void main(String[] args) {

Broker broker = new SimpleBroker();

StockExchange datafeed = new LondonStockExchangeAdapter();

AlgoTrader algoTrader = new AlgoTrader(broker, datafeed);

algoTrader.watchStock("GOOG");

algoTrader.watchStock("MSFT");

algoTrader.watchStock("APPL");

algoTrader.start();

}

(Broker.java)

package ic.doc;

public interface Broker {

void buy(String stock);

void sell(String stock);

}

(SimpleBroker.java)

public class SimpleBroker implements Broker {

...

}

2di)

(AlgoTraderTest.java)

@Rule public JUnitRuleMockery context = new JUnitRuleMockery();

final Broker broker = context.mock(Broker.class);

final StockExchange datafeed = context.mock(StockExchange.class);

AlgoTrader algoTrader = new AlgoTrader(broker, datafeed);

@Test

public void lowerPriceTellsBrokerSell() {

context.checking(new Expectations() {

{

oneOf(datafeed).currentPriceFor("GOOG");

will(returnValue(50));

oneOf(datafeed).currentPriceFor("GOOG");

will(returnValue(20));

oneOf(broker).sell("GOOG");

}

});

algoTrader.watchStock("GOOG");

algoTrader.trade();

algoTrader.trade();

}

2dii)

@Test

public void higherPriceTellsBrokerBuy() {

context.checking(new Expectations() {

{

oneOf(datafeed).currentPriceFor("GOOG");

will(returnValue(20));

oneOf(datafeed).currentPriceFor("GOOG");

will(returnValue(50));

oneOf(broker).buy("GOOG");

}

});

algoTrader.watchStock("GOOG");

algoTrader.trade();

algoTrader.trade();

}