1a

Builder - Search class is a builder for SearchQuery. I identified it by the build() method, which returns an instance of SearchQuery and the typical usage of method chaining to modify search parameters. Builders are used when the constructor of class has many parameters and we want to use some default values (either changeable or not).

Singleton - class RealEstateSite is constructed using the singleton pattern. I identified it by the usage of a call to RealEstateSite.getInstance() (instead of, for example, constructing a new instance). Singletons are used when there can only be a single instance of a class.

Factory - Price class instances are created using the factory pattern, identifiable by the usage of the static method Price.perWeekGbp() which returns a (presumably) new instance of the Price class. Factory pattern can be used to give a more idiomatic name to the class’s constructor, or if we need to defer the decision of what type of an object is needed until runtime.

1b

Singletons make testing more difficult, because they couple the singleton class and clients using it. It makes it impossible to test the client class without also testing the singleton. If the singleton is for example a database, we want to avoid making any changes to it during testing.

1c

Instead of acquiring the singleton instance of RealEstateSite directly in the class LondonPropertySearch we could pass in the constructor. This allows us to easily mock the RealEstateSite class.

public interface EstateAgent {

public List<Property> search (search(String postcode,

String city, String country);

}

public class RealEstateSite implements EstateAgent {...}

public class LondonPropertySearch {

private EstateAgent estateAgent;

public LondonPropertySearch(EstateAgent estateAgent) {

this.estateAgent = estateAgent;

}

//the rest of the class with the omission of

//RealEstateSite estateAgent = RealEstateSite.getInstance();

}

public class TestLondonPropertySearch {

public Mockery context = new Mockery();

private final EstateAgent estateAgent = context.mock(EstateAgent.class);

private final LondonPropertySearch lps = new LondonPropertySearch(estateAgent);

public void testSearchByPostcode() {

String postcode = “W12”;

context.checking( {

oneOf(estateAgent).search(

postcode, with(any(String.class)), with(any(String.class)))

});

lps.searchByPostcode(postcode);

}

public void testSearchByPrice() {

List<Property> fakeListings = Array.asList([

new Property(Price.perWeekGbp(400)),

new Property(Price.perWeekGbp(800)),

new Property(Price.perWeekGbp(300))]);

List<Property> expectedResult = new ArrayList();

expectedResult.append(fakeListings.get(0));

context.checking( {

exactly(1).of(estateAgent).search(“SW1”, “London”, “United Kingdom”);

will(returnValue(fakeListings);

});

List<Property> result = lps.searchByPrice(Price.perWeekGbp(350),

Price.perWeekGbp(600));

assertTrue(listsEqual(expectedResult, result));

}

}

Code is a bit shoddy, I can’t remember all the syntax atm (which is not the point of this exam anyway).

2a

i).

RMM is used to describe and categorise webservices based on how they use URIs and HTTP methods to serve requests. It has 4 levels (0-3) where level 3 services are fully REST-ful

ii).

Level 0: All requests are served on a single URL and use only the HTTP’s POST method with a payload identifying the actual resource or action required.

Level 1: Uses many URIs to represent different resources, however it doesn’t fully utilize all HTTP verbs. It will typically only use GET, which may side effect as well as return values.

Level 2: Uses many different URIs identifying the resource and makes use of various HTTP request methods (GET, POST, PUT, DELETE) to read or modify resources

Level 3: All the features of level 2 plus its response contains URI that points to other resources that client can use.

2b

i)

Model: Stores data, information representations and core business logic.

View: Takes information from the model and displays/renders it in some form.

Controller: Manipulates/updates the model

The MVC model is effective for applications with consistent core logic and multiple different

Views (i.e. a mobile app, a web app etc). With MVC, the model can be the source of data

for multiple different views. MVC allows for better separation of concerns and reduced code duplication.

ii) PAC may be favoured when there is a tree of dependencies between objects that each use their own MVC architecture. E.g. an online form that allows you to only click submit when all fields have been filled in. (A field will communicate up the tree to the submit buttons controller to signal to the button that the form has been filled in. The submit buttons modele will eventually update the view when each field is filled in.)

2c

class ImageEnhancerApplication {

private List<Image> images;

private JButton enhanceButton;

private Spinner spinner;

void initializeUI() {

spinner = new GuiSpinner();

enhanceButton = new JButton("Enhance");

enhanceButton.addActionListener(new Controller());

// rest of view

}

void enhance(Image image) {

// enhance

}

// Model

void enhanceImages() {

spinner.startSpinning();

ExecutorService executor = Executors.newFixedThreadPool(4);

CountDownLatch latch = new CountDownLatch(images.size());

for (Image image: images) {

executor.submit(new LatchedTask(latch, image));

}

latch.await();

spinner.stopSpinning();

}

private class LatchedTask implements Runnable {

private CountDownLatch latch;

private Image image;

public LatchedTask(CountDownLatch latch, Image image) {

this.latch = latch;

this.image = image;

}

@Override

public void run() {

try {

enhance(image);

latch.countDown();

} catch (InterruptedException e) { }

}

}

// Controller

class Controller implements ActionListener {

@Override

public void actionPerformed(ActionEvent e) {

enhanceImages();

}

}

}