Question 1 of 7

Consider the following code:

```
public class Cake {
   private double sugar;   //cup
   private double butter;  //cup
   private int eggs;
   private double flour;   //cup

   public Cake(double sugar, double butter, int eggs, double flour) {
      this.sugar = sugar;
      this.butter = butter;
      this.eggs = eggs;
      this.flour = flour;
   }

   public double getSugar() {
      return sugar;
   }

   public void setSugar(double sugar) {
      this.sugar = sugar;
   }

   public double getButter() {
      return butter;
   }

   public void setButter(double butter) {
      this.butter = butter;
   }
```

```
    public int getEggs() {
        return eggs;
    }

    public void setEggs(int eggs) {
        this.eggs = eggs;
    }

    public double getFlour() {
        return flour;
    }

    public void setFlour(double flour) {
        this.flour = flour;
    }
}
```

The problem with this code is that if you instantiate a cake object, the code looks like this:

```
Cake cake = new Cake(1.25, 1, 3, 3);
```

From this code it is not clear what the values of the arguments in the constuctor mean and it is easy to make mistakes with them.

Another problem is that this Cake class is mutable.

Rewrite this Cake class so that

1. The package class is immutable

2. We can create a package class with self-explaining code so that it is clear from the

code what the numbers 1.25, 1, 3, 3 mean.

Write both the code of the Cake class, and the code that creates a cake with sugar=1.25 cups, butter=1 cup, eggs=3, flour=3 cups.

```
// code of the Cake class:

public class Cake {

 // all instance variables are immutable to guarantee the Cake class is immutable

 private Double sugar;

 private Double butter;

 private Integer eggs;

 private Double flour;

 public Cake (Builder builder) {

  this.sugar = builder.sugar;

  this.butter = builder.butter;

  this.eggs = builder.eggs;

  this.butter = builder.flour;

 }
```

```
public class Builder {

  private Double sugar;

  private Double butter;

  private Integer eggs;

  private Double flour;

  public Builder withSugar (Double suagr) {

  this.sugar = sugar;

  return this;

}

public Builder withButter (Double butter) {

  this.butter = butter;

  return this;

}

public Builder withEggs (Integer eggs) {

  this.eggs = eggs;
```

```
    return this;

  }

  public Builder withFlour (Double flour) {

    this.flour = flour;

    return this;

  }

  public Cake build () { return new Cake(this); }

  }

}

// code to create a cake (not showing the whole code of the client class):

Cake cake = Cake.Builder

        .withSugar(1.25)

        .withButter(1.0)

        .withEggs(3)

        .withFlour(3.0)
```

```
        .build();
```

Question 2 of 7

Consider the following code:

```
public interface IVehicle {
    void start();
}

public class Car implements IVehicle {
   private String name ="Herbie";

   public void start() {
      System.out.println("Car " + name + " started");
   }

}

public class Logger implements InvocationHandler {
   private Object v;

   public Logger(Object v) {
      this.v = v;
   }

   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
      System.out.println("Logger: " + m.getName());
      Object object= m.invoke(v, args);
      return object;
```

```
    }
}

public class Notifier implements InvocationHandler {
    private Object v;

    public Notifier(Object v) {
        this.v = v;
    }

    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
        System.out.println("Notifier: " + m.getName());
        Object object= m.invoke(v, args);
        System.out.println("Notifier: " + m.getName());
        return object;
    }
}
```

What is the output written to the console when we run the following application?

```
public class Application {
    public static void main(String[] args) {
        IVehicle c = new Car();
        ClassLoader cl = IVehicle.class.getClassLoader();
        IVehicle v1 = (IVehicle) Proxy.newProxyInstance(cl, new Class[]
                { IVehicle.class }, new Logger(c));
        IVehicle v2 = (IVehicle) Proxy.newProxyInstance(cl, new Class[]
                { IVehicle.class }, new Notifier(v1));
        v2.start();
```

```
    }
}

// Console output:

Notifier: start

Logger: start

Car Herbie started

Notifier: start
```

Question 3 of 7

Consider the following code of a connection pool:

```
public class ConnectionPool {
    // this is a pool with only 1 connection
    private Connection connection = new Connection();

    public Connection getConnection() {
        return connection;
    }
}
```

For simplicity this ConnectionPool has only 1 connection.

Rewrite the ConnectionPool class so the the ConnectionPool class

1. is a singleton class

2. is reflection safe

3. is thread safe

4. gives the best performance in a multi threaded environment.

// Singleton code:

```
public class ConnectionPool {

private static Connection connection; // lazy-loading singleton

  private ConnectionPool() {

  // to make it reflection-safe

    if (connection!= null) {

      throw new RuntimeException("Use only the 'getConnection()' static method to get a singleton connection instance.");

    }

  }

  public static Connection getConnection() {

    // double-check locking pattern to make it thread-safe with best performance

    if (connection == null) {

      synchronized (ConnectionPool.class) {
```

```
        if (connection == null) {

          connection = new Connection ();

        }

       }

     }

    return connection;

  }

}
```

Question 4 of 7

Explain clearly the difference between the factory method pattern and the abstract factory pattern

The factory method pattern creates instances of related objects that implement the same interface. The factory here is a class that holds a reference to an interface/abstract class that is used to instantiate different implementations of the target class(es)/interfac(es).

Abstract factory pattern (also known as "Factory of factories") holds a reference to an interface of a factory class, which is used to create families of related objects by setting the factory reference to the required implementation, which instantiates the required "family" of products. For example, Production implementations or Mock implementations, etc.

In short: In the factory method pattern we instantiate the target class(es) directly, while the abstract factory pattern we instantiate a factory, which is used to instantiate the target class(es).

Question 5 of 7

Suppose we want to write our own Spring like framework so that the following application will work using the framework:

```
public class Application implements Runnable{
   @Inject
   BankService bankService;

   public static void main(String[] args) {
      FWApplication.run(Application.class);
   }

   @Override
   public void run() {
      bankService.deposit();
   }
}

public interface BankService {
   public void deposit() ;
}

@Service
public class BankServiceImpl implements BankService{
   @Inject
   private EmailService emailService;

   public void setEmailService(EmailService emailService) {
      this.emailService = emailService;
   }
```

```
    public void deposit() {
        emailService.send("deposit");

    }

}

public interface EmailService {
    void send(String content);
}

@Service
public class EmailServiceImpl implements EmailService{

    public void send(String content) {
        System.out.println("sending email: "+content);
    }
}
```

The framework will instantiate all classes annotated with @Service, and the framework supports dependency injection.

The code of the framework is as follows:

```
public class FWContext {

    private static List<Object> objectMap = new ArrayList<>();

    public FWContext() {
```

```
    try {
        // find and instantiate all classes annotated with the @Service annotation
        Reflections reflections = new Reflections("");
        Set<Class<?>> types = reflections.getTypesAnnotatedWith(Service.class);
        for (Class<?> implementationClass : types) {
            objectMap.add((Object) implementationClass.newInstance());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    performDI();
}

private void performDI() {
    try {
        for (Object theTestClass : objectMap) {
            // find annotated fields
            for (Field field : theTestClass.getClass().getDeclaredFields()) {
                if (field.isAnnotationPresent(Inject.class)) {
                    // get the type of the field
                    Class<?> theFieldType =field.getType();
                    //get the object instance of this type
                    Object instance = getBeanOftype(theFieldType);
                    //do the injection
                    field.setAccessible(true);
                    field.set(theTestClass, instance);
                }
            }
        }
    } catch (Exception e) {
```

```
            e.printStackTrace();
        }

    }

    public Object getBeanOftype(Class interfaceClass) {
        Object service = null;
        try {
            for (Object theTestClass : objectMap) {
                Class<?>[] interfaces = theTestClass.getClass().getInterfaces();

                for (Class<?> theInterface : interfaces) {
                    if (theInterface.getName().contentEquals(interfaceClass.getName()))
                        service = theTestClass;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return service;
    }

}

@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {
}

@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
public @interface Service {

}

public class FWApplication {

   public static void run(Class applicationClass) {
      // create the context
      FWContext fWContext = new FWContext();

      try {
        // you have to write the missing code

      } catch (Exception e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }
}
```

Write the missing code in the run() method of the FWApplication class.

You can write pseudo code as long as all steps of what the code should do is clear.

```
// pseudo-code for the missing part:

/*

  if(applicationClass instanceof Runnable) {
```

```
   // invoke the run() method.


 }


*/
```

Question 6 of 7

Explain clearly why all spring beans should have an interface if we want to make use of AOP in Spring.

In order to implement AOP in Spring, all Spring beans must have an interface because Spring AOP is based on the proxy design pattern. In this pattern, the proxy implements the same interface as the target class. So, instead of calling the target class's methods on the original object, they are called on the proxy. The proxy adds whatever logic required (whether before, after, or around the target method), calls the method on the original object, then returns the result.

The scenario above could never take place if the target object (the Spring bean in our case) did not implement an interface.

Question 7 of 7

Describe how Aspect Oriented Programming relates to one or more of the SCI principles you know. Your answer should be about half a page, but should not exceed one page (handwritten). The number of points you get for this question depends on how well you explain the relationship between Aspect Oriented Programming and the principles of SCI.

AOP is a manifestation of the SCI principle (Do less, accomplish more) because it focuses on addressing the cross-cutting concerns. Without AOP, if we (for example) wanted to log ten methods, we would have had to write the logging code ten times. With AOP, however, we can do that by writing the logging code in one place. Moreover, we can log whatever methods we want in the future (that is, accomplish more and more) with the same Aspect we wrote at the first time. So, as we log more objects, we accomplish more, and the effort that we did (writing the Aspect) becomes (compared to the accomplished tasks) less and less.

AOP also relates to the SCI principle of (Life exists in layers) because the AOP adds another layer that takes place before/after/around the original class(es). This layer adds intelligence to the original layers.

Also, we can see the SCI principle of (The whole is greater that the sum of its parts) when we implement AOP because the aspect class alone does nothing (its logic is meaningless without relating to another class), and the original classes do only their responsibilities. However, when the "Weaving" takes place, we end up with a new entity that does both the logic of the original/target class, and whatever logic written in the Aspect class. Also, if we wrote one Aspect class, and (for example) ten Target classes, we have eleven logic pieces, but in execution, twenty logic pieces are run (one for each time the advice method is run + one for running each JoinPoint).

AOP also relates to the SCI principle (Pure Consciousness is the source of all intelligence) because AOP implements a dynamic proxy, which depends heavily on the Reflections API. This API is analogous to the layer of pure consciousness because it contains the intelligence to get anything we need about any class, method, field, interface, or object instance while in runtime. It also contains the intelligence to act on them (invoke a method, for example). In other words, the Reflection API is the source of all intelligence of AOP, as Pure Consciousness is the source of all intelligence in life.

Also, the SCI principle (Every action has a reaction) is closely related to AOP: the action is calling any method that fulfills the criteria in the PointCut expression. The reaction is invoking the logic of the advice method.