[20 minutes]

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.

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

  public static class Builder {
     private double sugar;   //cup
     private double butter;  //cup
     private int eggs;
     private double flour;   //cup

  public Builder withSugarInCups(double sugar) {
     this.sugar = sugar;
     return this;
  }
  public Builder withButterInCups(double butter) {
     this.butter = butter;
     return this;
  }
  public Builder withNumberOfEggs(int eggs) {
     this.eggs = eggs;
      return this;
  }
  public Builder withFlourInCups(double flour) {
     this.flour = flour;
     return this;
  }

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

  public String toString() {
     return "Cake [sugar=" + sugar + ", butter=" +
butter + ", eggs=" + eggs + ", flour=" + flour + "]";
     }

  public Cake(Builder builder) {
     this.sugar = builder.sugar;
     this.butter = builder.butter;
```

```
        this.eggs = builder.eggs;
        this.flour = builder.flour;
    }

    public double getSugar() {
        return sugar;
    }

    public double getButter() {
        return butter;
    }

    public int getEggs() {
        return eggs;
    }

    public double getFlour() {
        return flour;
    }
}

public class ApplicationCake {

        public static void main(String[] args) {
                Cake cake = new Cake.Builder()
                                .withFlourInCups(3.5)
                                .withButterInCups(1.0)
                                .withNumberOfEggs(3)
                                .withSugarInCups(1.5)
                                .build();
                System.out.println(cake);
        }
}
```

[10 minutes]

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();
  }
}
```

```
Notifier: start
Logger: start
Car Herbie started
Notifier: start
```

[25 minutes]

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.

```
public class ConnectionPool {
  private static ConnectionPool pool;
  // this is a pool with only 1 connection
  private Connection connection = new Connection();
  private ConnectionPool() {
    // Prevent form the reflection api.
    if (pool != null) {
      throw new RuntimeException("Use getInstance() method to get the single
instance
                                 of this class.");
    }
  }
  public static ConnectionPool getPool() {
    // Double check locking pattern
    if (pool == null) { // Check for the first time
      synchronized (ConnectionPool.class) { // Check for the second time.
        if (pool == null)  pool = new ConnectionPool();
      }
    }
    return pool;
  }
  public Connection getConnection() {
    return connection;
  }
}
```

[10 minutes]

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

The factory method pattern is used to create one object and the abstract factory pattern is a factory of factories that is used to create a family of related objects

[30 minutes]

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.

```
// create instance of the application class
Object applicationObject = (Object)
applicationClass.newInstance();
// find annotated fields
for (Field field :
applicationObject.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 = fWContext.getBeanOftype(theFieldType);
      // do the injection
      field.setAccessible(true);
      field.set(applicationObject, instance);
  }
}
//call the run() method
if (applicationObject instanceof Runnable)
  ((Runnable)applicationObject).run();
```

[10 minutes]

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

Because Spring implements AOP using a proxy. This means the client that would normally talk to the target object will now talk to the proxy object in front of the target object. So in the code of the client we need to use the interface of the target class, otherwise we get an exception.

[10 minutes]

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.