Question 16 - Testing (Type 1, MP3)

Discuss the different types of testing that we have reviewed. What is the purpose of a Unit Test? What should you Unit Test? What is the purpose of an integration test? What is the purpose of using "mock" objects in our unit tests? Consider the articles assigned for reading in your answer.

A unit test is used to verify a single minimal unit of source code. The purpose of unit testing is to isolate the smallest testable parts of an API and verify that they function properly in isolation. - API Design for C++, p295; Martin Reddy

Unit testing purpose is to find and pinpoint the exact location of a mistake in a program. Some people have the notion that graceful error handling and logging will erases any need for unit testing. While error handling and logging helps us identify what happened and what to do when an error occurs, unit testing helps identify where it happened and what needs to be fixed so the error does not occur. Unit testing triggers those bugs in the testing phase before releasing to production. It ensures the behavior of the software is the intended behavior of the developer in addition to increasing the robustness of the software by gracefully handling errors. This means properly unit tested software it won't crash or operate on invalid data.

So how does one write a unit test? The first case one should write a unit test for is the common case (expected output), if there is one. Take this example:

```
public class CalulatorTest{

    private Calculator c;

    public CalculatorTest{
        c = new Calculator();
    }

    @Test
    public void testAdd(){
        assertEquals(c.Add(2,3), 5); /*<-- Common case, adding 2 numbers expected result*/
        assertEquals(c.Add(-1,null), "Error can't add null."); /*<-- Edge case, expects a string error message*/
        assertEquals(c.Add("%S",1), "Error can't add string to integer."); /*<-- Edge case, expects a string error message*/
    }
}
```

The common case will tell you if the code breaks, meaning you are getting unexpected output/results, after making some changes. Next, write tests for the edge cases, these cases could account for things like other expected output, null input, unexpected input, etc. From those unit test you can continue adding edge cases for things like whitespaces, unreasonably large numbers, so on and so forth. The beauty of unit testing is that if you find a bug while debugging you can add an edge case for that bug to the already existing unit test. After accounting for the newly debugged edge case in your unit test, one should only then proceed to correct the bug in the source code. Implementing unit testing increases confidence about the code written, and refactors to the code. It makes tracking down a bug much simpler and minimizes the time spent debugging complex systems.

Integration testing - Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. – International Software Testing Qualifications Board (ISTBQ).

Based on this definition the main purpose of an integration test is ensure that all components of the software work together seamlessly, and without error. There are several ways one could approach Integration testing:

- *Big Bang* is an approach to Integration Testing where all or most of the units are combined together and tested at one go. This approach is taken when the testing team receives the entire software in a bundle.
- *Top Down* is an approach to Integration Testing where top level units are tested first and lower level units are tested step by step after that. Test Stubs are needed to simulate lower level units which may not be available during the initial phases.
- *Bottom Up* is an approach to Integration Testing where bottom level units are tested first and upper level units step by step after that. Test Drivers are needed to simulate higher level units which may not be available during the initial phases.
- *Sandwich/Hybrid* is an approach to Integration Testing which is a combination of Top Down and Bottom Up approaches.

This gif beautifully illustrates the importance of integration testing http://giphy.com/gifs/unit-test-integration-3o7rbPDRHIHwbmcOBy. While unit tests make sure things work as expected, integration tests make sure things work TOGETHER as expected.

The purpose of using mock objects in unit tests is to help us isolate units. Isolating a unit from other objects means that the test will fail only if the unit fails. A mock object is simply a debug replacement for a real-world object. There are a number of situation in where mock objects can help us:

- The real object has nondeterministic behavior (Unpredictable results such as stock values)
- The real object is difficult to set up, or slow.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to confirm that a callback function was actually called).
- The real object does not yet exist (common problem when interfacing with other teams).

Overall mock objects allow us to continue to test units that have not been fully implemented by providing a “fake object” of what is actually expected. With this fake or “Mock” object we can test unit’s functionality as if it were fully implemented. Mock objects are used in a test environment to enable faster, more efficient testing.

References:

http://www.codeproject.com/Articles/727053/The-Purpose-of-a-Unit-Test
http://stackoverflow.com/questions/11917323/what-is-the-point-of-unit-testing
http://softwaretestingfundamentals.com/integration-testing/
http://media.pragprog.com/titles/utj/mockobjects.pdf

Question 1 - Java EE Specification and Platform (Type 1, Goncalves Chapter 1, Java EE Tutorial)

Discuss the Java EE specification and platform. Discuss the JCP and its relationship to the Java EE platform. Discuss the role of application servers as they relate to the platform, and what role the Glassfish application server plays. Discuss the different Java EE APIs and specifications. Discuss how Java EE relates to Java SE.

Java EE (Enterprise Edition) appeared towards the end of the 90's and brought the Java programming language to enterprise development. Platform specific tools include things like Enterprise Java Beans (EJB's), Hibernate validator, OpenMQ, etc. One of the main strengths of Java EE is that applications written with JPA, CDI, Bean Validation, EJBs, JSF, JMS, SOAP web services, or RESTful web services become portable across application servers like Glassfish or Tomcat. This sort dynamic behavior introduces an element of simplicity to creating enterprise solutions.

The JCP is an open organization, created in 1998 by Sun Microsystems, that is involved in the definition of future versions and features of the Java platform. When the need for standardizing an existing component or API is identified, the initiator (a.k.a. specification leader) creates a JSR (Java Specification request) and forms a group of experts. The group is composed of company representatives, organizations, universities, and private individuals. It becomes that group's responsibility to do the following 3 things:

- One or more specifications that explain the details and define fundamentals of the JSR
- A Reference Implementation (RI), which is an actual implementation of the specification
- Provide a compatibility Test Kit which is a set of tests every implementation needs to pass before claiming to conform to the specification.

When the JSR is approved by the executive committee (EC), the specification is released to the community for implementation.

In Java application servers, the server behaves like a virtual machine for running applications, handling connections to the database on one side, and connections to the Web client on the other. Application servers are the service layers of enterprise application. The purpose of the Glassfish server is that it's already preconfigured to run and host Java enterprise applications. Java EE itself acts as an abstract API which anyone can implement. Glassfish is a full implementation of Java EE having all the features and specifications built in. There are partial implementations like Tomcat which don't have all the features of Java EE but enough for a basic Java environment.

Java EE includes several API specifications, such as RMI, e-mail, JMS, web services, XML, etc., and defines how to coordinate them. Java EE also features some specifications unique to Java EE for components. These include Enterprise JavaBeans, connectors, servlets, JavaServer Pages and several web service technologies. This allows developers to create enterprise applications that are portable and scalable, and that integrate with legacy technologies. A Java EE application server can handle transactions, security, scalability, concurrency and

management of the components it is deploying, in order to enable developers to concentrate more on the business logic of the components rather than on infrastructure and integration tasks.

Java EE was built on top Java SE platform, which means Java SE APIs can be used in Java EE. Java Standard edition contains the core libraries and functions in Java. Java Enterprise edition took that and added libraries that introduced the ability to create large-scale, multi-tiered, scalable, reliable, and secure network applications.

Resources:
https://blogs.oracle.com/theaquarium/entry/java_ee_7_glassfish_4
http://everything.explained.today/Java_Platform,_Enterprise_Edition/

Question 5 - Java EE Concepts and PaaS (Type 1, MP1, Goncalves Chapter 1, Java EE Tutorial)

Explain the general benefits associated with enterprise applications that have multiple software layers. Consider concepts like scalability, performance and software development practices in your answer. How might the size of an application, or the size of a development team, relate to an multi-layered architecture? Discuss the role of PaaS as it applies to modern development practices. Discuss the different layers or tiers in the Java EE programming model. Discuss the use of Declarative Programming, Inversion of Control and Configuration by Exception in the Java EE platform.

The general benefits of associated with multi-layered enterprise applications are scalability, maintainability, reliability, availability, extensibility, performance, manageability, and security. The multi-layer concept provides flexibility and structure, making enterprise apps easier to develop and maintain. According to our textbook Java EE has 3 layers requiring validation:

- Presentation layer: In this layer you validate the data because the data could have been sent from several clients (a web browser, a command line tool such as cURL, which allows you to send HTTP commands, or a native application). You also want to give your users a quicker feedback. (Presentation layer aka view in MVC)
- Business (application) logic layer: This layer orchestrates the calls to your internal and external services, to your domain model so the processed data have to be valid. (Persistence layer or controller in MVC)
- Business model layer: This layer usually maps your domain model to the database, so you need to validate it before storing data. (Business layer or model in MVC)
  - Goncalves P.68
  -

This distribution of these layers allow developers to work on a layer individually without affecting the other layers. It also allows developers to scale the application by the layers. Overall, the layer model increases performance by grouping related tasks into layers.

The size of an application can relate to the number of layers it might have. If you're developing a single donation landing page you could do everything simply on one layer.

However, if you wanted to create a scalable enterprise donation system it would have multiple layers. The larger the app the more layers it will contain since it needs to be easily scalable, maintainable, extensible, manageable and secure. The size of the development team would correspond with the size of the app. For example, the single donation landing page could be done on a single layer by just one developer. For an enterprise donation system it might require a single team of 3-5 developers per layer, or even multiple teams per layer depending on the complexity of the apps presentation, business (application) logic, and business model.

Platform as a service (PaaS) role in modern development practices today is to eliminate the time needed to setup the server platform so that web applications can be developed and deployed almost instantly. The executive summary of PaaS on Engineyard, a PaaS provider with clients such as FOX, and MTV states "Rather than downloading and building all of those platform-level technologies on each server instance, and then later having to repeat the process as you scale, you can go to a simple Web user interface, click a few options, and have your application automatically deployed to a fully provisioned cluster" (Engineyard Para. 2). This is exactly what we did with OpenShift in MP1 and it saved us time.

The use of declarative programming in Java EE7 is necessary in order to create the enterprise applications the platform is intended for. The declarative style allows us to focus more about the business problems at hand by allowing us to tell the computer what needs to be done, not how to do it. Examples of this in Java EE7 would be operations performed on the database (modal) in SQL. This use of declarative programming would likely be implemented in the business logic layer or even in the business modal layer. The Java programming language however is imperative meaning you tell the computer how to do it. For example the doPost() method in Java EE MVC is where you state what to do when a post back occurs. In MP2 I imperatively created a Customers object in the doPost() method, explicitly telling Java how to create the object.

Inversion of control is all about removing dependencies from your code by increasing modularity making the program more extensible. For example in MP2 we added hibernate validation as a dependency in our pom.xml so that our web app didn't need to depend on a validation class in our project. Using Java EE7 we can directly inject injection @Resource we are able to use hibernates validation features without having to ever create a validation object.

IoC Example:

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-validator</artifactId>
        <version>5.2.4.Final</version>
    </dependency>
</dependencies>
```

```
@WebServlet(name = "CustomerServlet", urlPatterns = {"/custom
public class CustomerServlet extends HttpServlet {

    @Resource
    Validator validator;
}
```

```
public class Customer {

    @NotNull
    @Min(value = 10, message="Customer ID must be inbetween 10 and 99999.")
    @Max(99999)
    private Long id;

    @NotNull
    @Size(max=45)
    private String firstName;

    @NotNull
    @Size(max=45)
    private String lastName;
```

As you can see in the CustomerServlet uses @Resource to create a validator from the xml dependency. From there the dev can declaratively implement the validation in the Customer class. IoC is very useful because allows the developer to focus solely on the business logic without adding clutter to the programs structure.

Configuration by Exception provides a way to externally configure your application at deployment time. The example given by Antonio Goncalves from his blog states: "If you deploy your application in Europe you need to use the Euro as the current currency and if you deploy it in the US you need dollars" (http://antoniogoncalves.org Para. 1). This sort of dynamic currency functionality could be achieved prior to deployment, but would be a nightmare to implement for each and every country in the world. This is where Configuration by Expectation comes into play. There are a number of ways one can configure their application. You can configure parameters within servlets in the web.xml, EJB's in the ejb-jar.xml, application specific info in the application.xml, and data source parameters in the persistence.xml. Configuring these files can give your application the dynamic behavior it requires, without having to implement anything extra prior to deployment.

Resources:

http://www.developer.com/design/article.php/10925_3808106_5/Introducing-Enterprise-Java-Application-Architecture-and-Design.htm
http://stackoverflow.com/questions/3441294/why-we-should-develop-a-application-with-multiple-layers
https://www.engineyard.com/whitepapers/top-10-advantages-of-platform-as-a-service/
http://java.boot.by/scea5-guide/ch02s03.html
http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart1.aspx
http://antoniogoncalves.org/2011/06/10/debate-and-what-about-configuration-in-java-ee-7/

For MP3 you are designing and developing your own domain model that will carry forward into MP4 and the Final Project.
For the purposes of answering this question the general requirements are:

1. **You must have at least six entities**
2. **Entities must support relationships between one another (OneToMany, ManyToMany, etc)**
3. **Your idea must support the logical concept of multiple types of users or roles. We must consider this now so we can introduce security in later projects.**

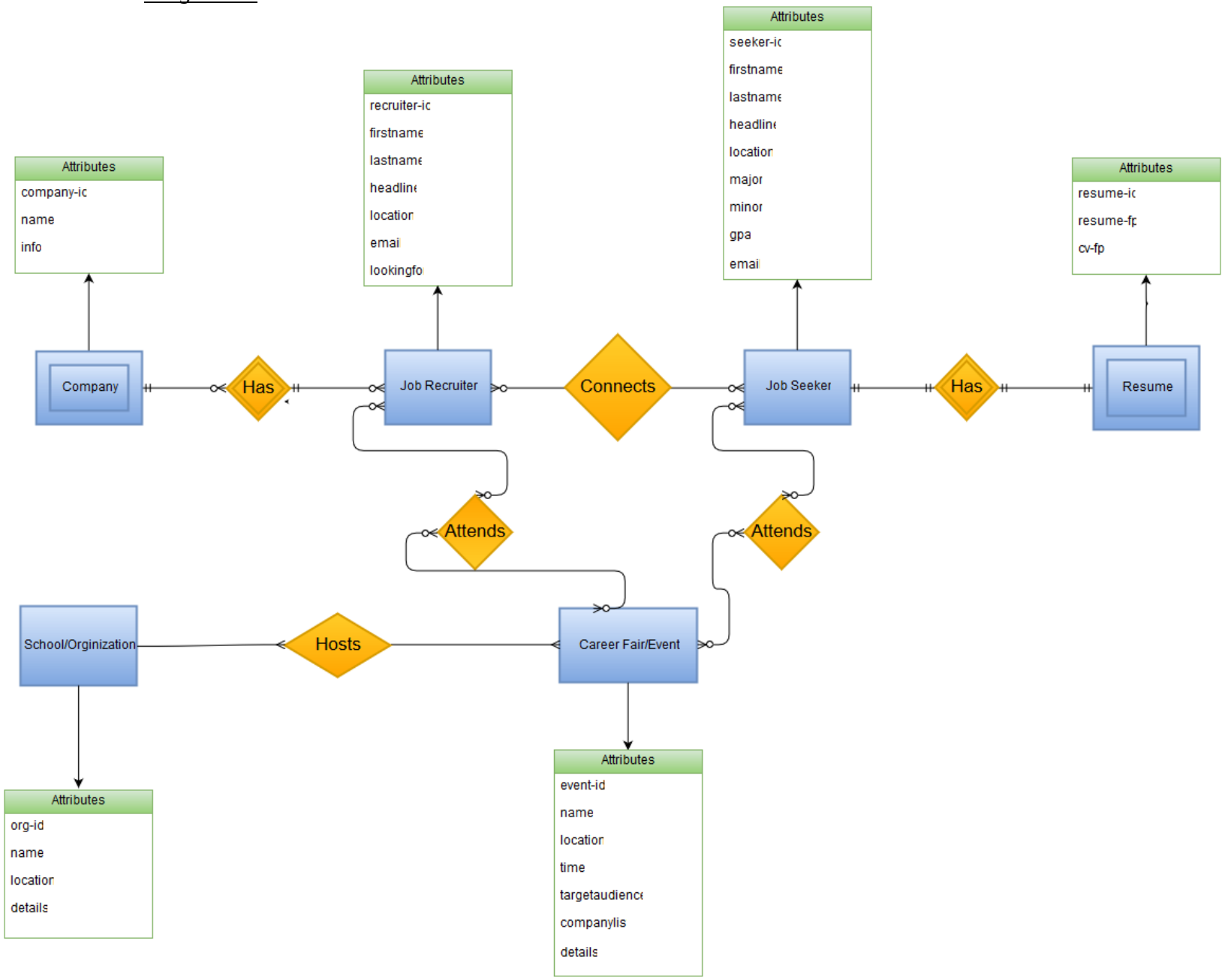
To receive full credit for this question, please make sure to:

1. **Discuss your Entities. Including a diagram is strongly encouraged.**
2. **Discuss how your Entities relate to one another, including any use of JPA inheritance strategies**
3. **Discuss what functionality you will expose in your FP web application, and how that functionality relates to your multiple roles/security. In other words, who can do what?**

For my final project I will be implementing my IPRO project from last semester – NEX. Screenshots can be found here: https://github.com/adeebahmed/NEX/tree/master/App. NEX is an app that aims to modernize the career fair/interview process via a tool (app) that allows job recruiters to seamlessly interact with job seekers by connecting with the person at career fairs/event.

Diagram 1 illustrates the relations between my 6 entities: JobSeekers, JobRecruiters, Organization, Event, and weak entities Company and Resume.

Diagram 1:



My entities are intended to relate as follows:
- A job recruiter needs a company, a company can have multiple job recruiters.
- Job recruiters can connect with job seekers, job seekers can only connect if recruiters allow it
- Job seekers have a single resume, resumes have a single job seeker
- Job seekers and job recruiters both attend an event (this is where they are intended to connect)
- An organization will host the event that job seekers and job recruiter can attend
- My modal has the following relations: one-to-one, one-to-many, and many-to-many

Functionality wise I intend this application to be used by job recruiters at various companies and job seekers at various schools/organizations. Job recruiters will be able to logon to the app from their end and customize a profile for their company and themselves for a specific career fair/event.

Prior to the event job seekers will be able to see company profile pages as well as job recruiter's page. The recruiter should provide details on the company and their page that will attract job seekers to them when they are at the event. From the job recruiter end they can post updates about job openings at their company and seekers can be notified easily about these updates. Job seekers will be able to also construct their own profile on the app and attach their resume. They can make their profile private if they do not wish to be on the job market.

When a recruiter and job seeker connect they will be able to message each other. If they are not connected only viewing profile recruiter and company profiles is allowed. Job seekers will open the app and search for career fair/events in the area based on their major and minor. They can discover events around them and the recruiters and companies that will be there all from one convenient app. The purpose of this app to create long and meaningful connections, rather than have people attended a resume give-away, just to be hit by 20 drive-by inbox messages.