Franklin Perez: Explanation of Strategy Design Pattern and Sorting Algorithms to Choose During 10/7/2011 Telephone Interview for Java Deveeloper Build Tools Disney Position

From: **Franklin Perez** (perezfranklin@hotmail.com)
Sent: Sat 10/08/11 1:30 PM
To: bob@bobrubano.com
Bcc: cdunn@softwareresources.com

Dear Mr. Bob Rubano:

My name is Franklin Perez, and I interviewed with you and Muhammad Yahia at 2:00 PM on 10/7/2011 over the phone for the Java Developer (with Build Tools) contracting position at Disney. I would very much like the opportunity to show how much of an asset I can be for your project at Disney.

During the 10/7/2011 telephone interview, Muhammad Yahia asked me about the Strategy Pattern and which Sorting Algorithms should be chosen based on a certian setof criteria. I would now like to explain the Strategy Design Pattern within the context of choosing the best Sorting Algorithm based on a certain set of criteria.... Please forward this to Muhammad Yahia.

The Strategy pattern is much like the State pattern in outline but a little different. The Strategy pattern consists of a number of related algorithms encapsulated in a driver class called Context. Your client program can select one of these algorithms, or in some cases Context might select the best one for you. The intent is to make these algorithms interchangeable and provide a way to choose the most appropriate one. The The State and Strategy differ in that the user generally chooses which of several strategies to apply, and only one strategy at a time is likely to be instantiated and active within the Context class. By contrast, all of the different states could possibly be active at once, and switching between them might often occur. In addition, Strategy encapsulates several algorithms that do more or less the same thing, while State encapsulates related classes that each do something somewhat different. Finally, the concept of transition between different states is completely missing in the Strategy pattern.

Below is some java pseudo-code explaining how to implement the Strategy Pattern within the context of choosing the best Sorting Algorithm given a set of criteria....

```
public abstract class SortStrategy {
   public abstract void sort(List aList);
}

public class BubbleSortStrategy extends SortStrategy {
   public sort void sort(List aList) {
      // Implementation code for Bubble Sort
   }
}

public class MergeSortStrategy extends SortStrategy {
   public void sort(List aList) {
      // Implementation code for Merge Sort
   }

public class HeapSortStrategy extends SortStrategy {
```

```
    public void sort(List aList) {
        // Implementation code for Heap Sort
    }
}

public class QuickSortStrategy extends SortStrategy {
   public void sort (List aList) {
      // Implementation code for Quick Sort
   }
}

public class SelectionSortStrategy extends SortStrategy {
   public void sort (List aList) {
       // Implementation code for Selection Sort
   }
}

public class InsertionSortStrategy exteends SortStrategy {
   public void sort(List aList) {
       // Implementation code for Insertion Sort
   }
}

public class BucketSortStrategy extends SortStrategy {
   public void sort(List aList) {
       // Implementation code for Bucket Sort
   }
}

public class SortContext {
   private SortStrategy aSortStrategy;

   // Why use Merge Sort as default Sort Strategy?
   // ---------------------------------------------------
   //
   // According to the "Comparing Sorting Algorithms" section of the book "Beginning Programming for
Dummies" and the
   // http://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms hyperlink:
   // 1) The three fastest sorting algorithms are heap sort, merge sort, and quick sort. Although quick sort is
considered the fastest of the three algorithms,
   //     merge sort is faster in worst-case scenarios.
   // 2) Also, the Heap Sort and Quick Sort algorithms are unstable.
   //
   // Also, according to the http://download.oracle.com/javase/tutorial/collections/algorithms/index.html
hyperlink, The Java platform uses a modified version of the
   // merge sort because it's fast and stable.
   //
   // Therefore, due to all the above reasons, the Merge Sort algorithm was chosen as the default Sort Strategy.
   //
   public SortContext() {
      setSortStrategy( new MergeSortStrategy() );
```

```
    }

  public SortContext(SortStrategy argSortStrategy) {
      if ( argSortStrategy != null ) {
          setSortStrategy(argSortStrategy);
      }
      else {
          setSortStrategy( new MergeSortStrategy() );
       }
  }

  public void setSortStrategy(SortStrategy argSortStrategy) {
     if ( argSortStrategy != null ) {
        aSortStrategy = argSortStrategy;
     }
  }

  // This method is used to let the SortContext class decide which SortStrategy should be used based on a set of
inputs from the "aMap" input.
  //
  // The input String keys that can be used inside the "aMap" are:
  // 1) "Only a Few Items on List": True or False
  // 2) "Items Are Mostly Sorted Already": True or False
  // 3) "Concerned about worst-case scenarios": True or False
  // 4) "Interested in a good average-case result": True or False
  // 5) "Items are drawn from a dense universe": True or False
  // 6) "List to sort": List object
  // 7) "Types if Objects in List": String value stating Object Type
  // 8) "Limited Memory Concerns": True or False
  // 9) "Stability Concerns": True or False
  // 10) "Interested in fastest average case": True or False
  //
  // What's exact criteria for choosing sort algorithm? It's a heuristic approach....
  //
  // According to the http://warp.povusers.org/SortComparison/ hyperlink:
  // 1) Insertion Sort: Insertion sort is good only for sorting small arrays (usually less than 100 items). In fact, the
smaller the array, the faster insertion
  //     sort is compared to any other sorting algorithm. However, being an O(n^2) algorithm, it becomes very slow
very quick when the size of the array
  //     increases.
  // 2) Heap Sort: Heap sort is the other (and by far the most popular) in-place non-recursive sorting algorithm....
Heap sort is
  //     not the fastest possible in all (nor in most) cases, but it's the de-facto sorting algorithm when one wants to
make sure that the sorting
  //     will not take longer than O(nlogn). Heap sort is generally appreciated because it is trustworthy: There
aren't any "pathological" cases
  //     which would cause it to be unacceptably slow. Besides that, sorting in-place and in a non-recursive way
also makes sure that it will
  //     not take extra memory, which is often a nice feature.
  // 3) Merge Sort: The virtue of merge sort is that it's a truely O(nl og n) algorithm (like heap sort) and that it's
stable (it doesn't change
```

```
    //     the order of equal items like eg. heap sort often does). Its main problem is that it requires a second array with the same size as
    //     the array to be sorted, thus doubling the memory requirements.
    // 4) Quick Sort: Quicksort is the most popular sorting algorithm. Its virtue is that it sorts in-place (even though it's a recursive algorithm) and that
    //     it usually is very fast. One reason for its speed is that its inner loop is very short and can be optimized very well.... The main problem with quicksort
    //     is that it's not trustworthy: Its worse-case scenario is O(n^2) (in the worst case it's as slow, if not even a bit slower than insertion sort) and
    //     the pathological cases tend to appear too unexpectedly and without warning, even if an optimized version of quicksort is used....
    //
    // According to Table 4-6 of the "Algorithms in a nutshell" book (and other sources), the Sorting algorithms recommended based on criteria are as follows:
    // 1) Criteria: Only a few items (like less than 100 items)
    //     Sorting Algorithm Recommended: Insertion Sort - According to the http://warp.povusers.org/SortComparison hyperlink: Insertion sort is
    //         good only for sorting small arrays (usually less than 100 items). In fact, the smaller the array, the faster insertion sort is compared to
    //         any other sorting algorithm. However, being an O(n^2) algorithm, it becomes very slow very quick when the size of the array increases.
    // 2) Criteria: Items are mostly sorted already
    //     Sorting Algorithm Recommended: Insertion Sort
    // 3) Criteria: Concerned about worst-case scenarios
    //     Sorting Algorithm Recommended: Heap Sort (Note: I would personally choose the Merge Sort due to stability reasons.)
    // 4) Criteria: Interested in a good average-case results
    //     Sorting Algorithm Recommended: Quick Sort
    // 5) Criteria: Items are drawn from a dense universe
    //     Sorting Algorithm Recommended: Bucket Sort
    // 6) Criteria: Limited Memory concerns
    //     Sorting algorithm recommended: Heap Sort - According to the http://warp.povusers.org/SortComparison hyperlink: Heap sort is the
    //         other (and by far the most popular) in-place non-recursive sorting algorithm.... Heap sort is not the fastest possible in all (nor in most)
    //         cases, but it's the de-facto sorting algorithm when one wants to make sure that the sorting will not take longer than O(nl og n).
    //         Heap sort is generally appreciated because it is trustworthy: There aren't any "pathological" cases which would cause it to be
    //         unacceptably slow. Besides that, sorting in-place and in a non-recursive way also makes sure that it will not take extra memory,
    //         which is often a nice feature.
    // 7) Criteria: Large Number of Items to Sort
    //     Sorting algorithm recommended: Merge Sort or Quick Sort - According to the
    //         http://math.hws.edu/TMCM/java/labs/xSortLabLab.html hyperlink: Merge Sort and QuickSort are more complicated, but also much faster for large lists.
    //         If stability is an issue also, then would choose Merge Sort. If Stability is not an issue and just interested in fastest average case sorting, then
    //         Quick Sort is chosen.
    //
    public void setSortStrategy(Map aMap) {
```

```
        // Implementation code that takes a set of inputs from the "aMap" input with key/value pairs and makes the
decision as to which SortStrategy will be
        // set based on those inputs. The comments above gives a heuristic approach in choosing which
SortingStrategy will be used.

        // If
    }

    public void sort(List aList) {
        aSortStrategy.sort(aList);
    }
}
```

A user would create an instance of the SortContext class and then use the SortContext class to sort a List of Items. If a user wants to just use the default, then the
following code would be used:

```
        List listOfItems = <some list that's been instantiated with items>
        SortingContext sortingContext = new SortingContext();
        sortingContext.sort(listOfItems);
```

If a user wants to explicitly choose the Sorting Algorithm to use. such as QuickSortStrategy, then the following can be done....

```
        List listOfItems = <some list that's been instantiated with items>
        SortingContext sortingContext = new SortingContext( new QuickSortStrategy() );
        // Could also use following:
        //     SortingContext sortingContext = new SortingContext();
        //     sortingContext.setSortStrategy( new QuickSortStrategy() );
        sortingContext.sort(listOfItems);
```

If the user is not sure what Sorting Strategy to use but wants to give the SortingContext instance some set of criteria, something as shown below can be done:

```
        List listOfItems = <some list that's been instantiated with items>
        SortingContext sortingContext = new SortingContext();

        Map inputArgs = new HashMap();
        inputArgs.put("Interested in fastest average case", new Boolean(Boolean.TRUE);
        inputArgs.put("Stability Concerns", new Boolean(Boolean.FALSE);

        sortingContext.sort(listOfItems);
```

Anyways, I hopes this demonstrates my knowledge of the subject that perhsp did not come across during the interview.

If you may please forward this to Muhammad Yahia, I would most greatly appreciate it. It will at least demonstrate that I put forth extra effort in finding a solution to the questions asked of me and is a better

demonstration of my knowledge of the subject.

And again, I believe I would be a good asset to your project. I bring a lot to the table as part of my experience and second effort in finding solutions to problems.

Thanks,

Franklin Perez
Cell: (407) 694-7805
Home: (407) 977-1419

From: perezfranklin@hotmail.com
To: bob@bobrubano.com
Subject: Franklin Perez: More Complete Answers to Questions During 10/7/2011 Telephone Interview for Java Deveeloper Build Tools Disney Position
Date: Sat, 8 Oct 2011 00:41:56 +0000

Dear Mr. Bob Rubano:

My name is Franklin Perez, and I interviewed with you and Muhammad Yahia at 2:00 PM on 10/7/2011 over the phone for the Java Developer (with Build Tools) contracting position at Disney. I would very much like the opportunity to show how much of an asset I can be for your project at Disney.

The following are my more complete answers to the questions that Muhammad Yahia asked me during the interview:

Question #1: What's the difference between Java 1.4 and Java 1.5?

Answer(s) to Question #1:

1) Support for Generics, which allows a type or method to operate on objects of various types while providing compile-time safety. Now, you can instantiate a Java Collection that can hold objects of a specific type. For example, you could have the following code that compiles just fine, but may cause a run-time error when adding a String to a ArrayList but then when getting on object from the ArrayList and casting it as an Integer causes a run-time error:

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer) v.get(0)  // Run-time error
```

Now... with use of Generics capability in Java 1.5, you can instantiate an ArrayList specifying the types of Objects that may be in it, and if you later try to get the object and cast it to some other type of object, you will now get a compile error, not a run-time error. Example follows:

```
List <String> v = new ArrayList<String>();   // Only objects of type String may be entered into the "v" ArrayList.
v.add("test");   // no compile error or problems
v.add( new DummyClass() ); // compile error; problem caught at compile time vs. run-time
Integer i = (Integer) v.get(0);  // Compile type error because it's a String that's being returned from v.get(0), not object. Problem caught at compile time.
```

The advantage of catching the problem at compile-time vs. run-time is that it requires much less time and effort

to fix such a problem. The problem is caught much sooner. This reduces debugging time immensely when the problem is caught at compile-time vs. run-time.

Also, now you can instantiate Lists of Strings, Integers, ClassA, ClassB, etc, without having to create a new Class that supports only those types of classes. Thus, making software programming easier and cutting down on amount of code.

2) The underlying virtual machine also includes a number of enhancements. The VM now loads all classes from a running JAR file to an internal, shared archive where they can be accessed by other processes running on the VM, improving efficiency for classes that are likely to be used by a wide variety of simultaneously running processes. Improvements have also been made to the efficiency of the garbage collector, thread priority mapping, and fatal error diagnostics.

3) The Remote Method Invocation has been enhanced. It now supports the dynamic generation of stub classes at runtime. SSL and TSL socket factory classes have been added to simplify communication over the SSL and TLS protocols. Enhancement have also been made to the Java Database Connectivity (JDBC) package, particularly in the RowSet interface, which provides an easy way to pass data from a database connection between components. There are five standard implementations. There are also new features for CORBA, Java IDL, Java RMI-IIOP, and the JNDI.

4) The base libraries have received a wide variety of enhancements. Some notables include: Lang and Util now have a Formatter class containing a variety of tools to help with common string formatting problems. The new scanner class can be used to convert text into primitives and strings, as well as conduct regular expression searches on files, streams, and other implementors of the Readable interface. A number of tools have been added to assist in concurrent and multithreaded programming.

Question #2: Composition vs. Inheritance - When would you use one over the other?

Answer(s) to Question #2:

1) In Java: If you want a class to have the characteristics of two classes. In Java, you can only have single-inheritance. So, if you want a class (ChildClass) to have the characteristics of two classes (ClassA and ClassB), you can extend from one class (say ClassA) via inheritance and have the other class (ClassB) as a data member of ChildClass in a has-a relationship. So... you'd do this....

```
public class ClassA {
   // ....
}

public interface ClassBInterface {
    public void classBmethodOne(....);
    public void classBmethodTwo(...);
}

public class ClassB implements ClassBInterface {
   public void classBmethodOne(....) {
      // ...
  }

  public void classBmethodTwo(...) {
     // ...
```

```
  }
}

public class ChildClass extends ClassA implements ClassBInterface {
   private ClassB classBinstance = new ClassB();

   public void classBmethodOne(....) {
      classBinstance.classBmethodOne(...);
  }

  public void classBmethodTwo(...) {
      classBinstance.classNmethodTwo(...);
  }
}
```

2) Use inheritance if you know for sure that a certain class truly is an is-a relationship. If the is-a relationship will be constant throughout the lifetime of the software, then use inheritance. For example, a `HouseOwner` is a *Person*; however, a `HouseOwner` is not a House. Thus, the `HouseOwner` class should not inherit from the *House* class but should inherit from the *Person* class because a *HouseOwner* is a *Person*.

3) Don't use inheritance just to get code reuse. If you want to just have code re-use and there is no real is-a relationship, then do not use inheritance. You should use composition.

4) If you have a class that has characteristics of two other classes that will not be permanent throughout the lifecycle of the software, then you should use composition. For example, you might think that an `Employee` is-a `Person`, when really `Employee` represents a role that a `Person` plays part of the time. What if the person becomes unemployed? What if the person is both an `Employee` and a `Supervisor`? Such impermanent is-a relationships should usually be modeled with composition.

5) Don't use inheritance just to get at polymorphism. If all you really want is polymorphism, but there is no natural is-a relationship, use composition with interfaces as follows:

```
public interface ClassBInterface {
    public void classBmethodOne(....);
    public void classBmethodTwo(...);
}

public class ClassA implements ClassBInterface {
   private ClassB classBInstance = new ClassB();

   public void classBmethodOne(....) {
      classBinstance.classBmethodOne(...);
  }

  public void classBmethodTwo(...) {
      classBinstance.classNmethodTwo(...);
  }
}
```

6) Use composition when you want to model has-a relationships. For example, it is natural to include an attribute referencing to *House* in the `HouseOwner` class, because a `HouseOwner` has a *House*. The has-a relationship

would be modeled as follows:

```
public class House {

}

public class HouseOwner  {
   private House aHouse;

  public HouseOwner(House house) {
     aHouse = house;
  }
}
```

Other things to keep in mind when using composition vs. inheritance....
1) It is easier to change the interface of a back-end class (composition) than a superclass (inheritance). A change to the interface of a back-end class necessitates a change to the front-end class implementation, but not necessarily the front-end interface. Code that depends only on the front-end interface still works, so long as the front-end interface remains the same. By contrast, a change to a superclass's interface can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclass's interface. (Note: The back-end class is the class that acts as a superclass but it's now a data member of the sub-class. This is how you can "subclass" from another class using composition.)
2) It is easier to change the interface of a front-end class (composition) than a subclass (inheritance). Just as superclasses can be fragile, subclasses can be rigid. You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its supertypes. For example, you can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
3) Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object. With inheritance, you get the image of the superclass in your subclass object image as soon as the subclass is created, and it remains part of the subclass object throughout the lifetime of the subclass.
4) It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition, *unless* you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
5) The explicit method-invocation forwarding (or delegation) approach of composition will often have a performance cost as compared to inheritance's single invocation of an inherited superclass method implementation. I say "often" here because the performance really depends on many factors, including how the JVM optimizes the program as it executes it.
6) With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remain inside the same class.

Question #3: You have a List of Integers. Which sorting method would you use if space/resources were limited?

Answer(s) to Question #3:

Normally, I would just use the Java Sort facility that's provided. I'll assume that I have a List of Integer objects that have already been put inside of a List. Example code shown below:

```
List<Integer> list = new ArrayList<Integer>
addIntegers(list);
Collections.sort(list)
```

Now, the "list" has been sorted. After doing some research at the http://download.oracle.com/javase/tutorial/collections/algorithms/index.html hyperlink, I found out the following:

> The `sort` operation uses a slightly optimized *merge sort* algorithm that is fast and stable:
>
> - **Fast**: It is guaranteed to run in `n log(n)` time and runs substantially faster on nearly sorted lists. Empirical tests showed it to be as fast as a highly optimized quicksort. A quicksort is generally considered to be faster than a merge sort but isn't stable and doesn't guarantee `n log(n)` performance.
> - **Stable**: It doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes. If a user of a mail program sorts the inbox by mailing date and then sorts it by sender, the user naturally expects that the now-contiguous list of messages from a given sender will (still) be sorted by mailing date. This is guaranteed only if the second sort was stable.

Therefore, after some reflection, I think that I would be inclined to use a merge sort algorithm due to the reasons stated above.

If there are any further questions you or Muhammad Yahia would like to ask me, please let me know, and I'll do my best to answer them.

If you may please forward this to Muhammad Yahia, I would most greatly appreciate it. It will at least demonstrate that I put forth extra effort in finding a solution to the questions that were asked of me.

And again, I believe I would be a good asset to your project. I bring a lot to the table as part of my experience and second effort in finding solutions to problems.

Thanks,

Franklin Perez
Cell: (407) 694-7805
Home: (407) 977-1419

From: perezfranklin@hotmail.com
To: bob@bobrubano.com
Subject: Franklin Perez: Number of Permutations for Seating N Guests in Round Table? (N-1)!
Date: Fri, 7 Oct 2011 21:05:37 +0000

Dear Mr. Bob Rubano:

My name is Franklin Perez, and I interviewed with you and Muhammad Yahia at 2:00 PM over the phone for the Java Developer (with Build Tools) contracting position at Disney. I would very much like the opportunity to show

how much of an asset I can be for your project at Disney.

I now remember how many permutations exists for seating "N" number of guests in a round table. According to the http://www.algebra.com/algebra/homework/Permutations/Permutations.faq.question.85120.html hyperlink, it would be "(N-1)!" The "!" symbol means factorial. The concept of factorial is explained in the http://en.wikipedia.org/wiki/Factorial hyperlink.

So, 5! would equal 5*4*3*2*1. So, if you were to try and seat 7 people, the number of possible seating arrangements would be (7-1)!, which would be 6*5*4*3*2*1 = 720.

We could use the brute force approach to view every possible permutation of seating arrangements until a seating arrangement comes up that has nobody seated to someone else they dislike.

If you may please forward this to Muhammad Yahia, I would most greatly appreciate it. It will at least demonstrate that I put forth extra effort in finding a solution to the problem.

And again, I believe I would be a good asset to your project. I bring a lot to the table as part of my experience and second effort in finding solutions to problems.

Thanks,

Franklin Perez
Cell: (407) 694-7805
Home: (407) 977-1419

From: perezfranklin@hotmail.com
To: bob@bobrubano.com
Subject: Franklin Perez: Found Solution to Guests Seating Arrangement Problem Presented by Muhammad Yahia
Date: Fri, 7 Oct 2011 20:31:17 +0000

Dear Mr. Bob Rubano:

My name is Franklin Perez, and I interviewed with you and Muhammad Yahia at 2:00 PM over the phone for the Java Developer (with Build Tools) contracting position. I would very much like the opportunity to show how much of an asset I can be for your project at Disney.

I found the solution to the following question that was asked of me by Muhammad Yahia: "We have invited *n* people to dinner. We have a big round table for the occasion, but we have a problem. Many of our guests don't get along with one another. Every dislike is mutual. Indeed, each person may have up to ½*n*-1 enemies in the group. **Write an algorithm that creates a seating arrangement in which no person sits next to an enemy.**"

I found the solution at the http://www.cs.uni.edu/~wallingf/teaching/153/sessions/session22.html hyperlink.

Below are the two that I found:

Solution #1:

```
seating = empty list
guests  = queue containing all guests
```

```
    until guests is empty
        g = guests.dequeue()
        if g will sit by rightmost guest in seating,
           seat g there
        else if g will sit by leftmost guest in seating,
           seat g there
        else
           guests.enqueue(g)
```

Solution #2:

```
 seating = a random permutation of the guests

    until all guests are happy
        work around the table,
        making a swap whenever we encounter a conflict
```

If you may please forward this to Muhammad Yahia, I would most greatly appreciate it. It will at least demonstrate that I put forth extra effort in finding a solution to the problem.

And again, I believe I would be a good asset to your project. I bring a lot to the table as part of my experience and second effort in finding solutions to problems.

Thanks,

Franklin Perez
Cell: (407) 694-7805
Home: (407) 977-1419

From: CDunn@softwareresources.com
To: perezfranklin@hotmail.com
Date: Thu, 6 Oct 2011 09:29:14 -0400
Subject: Disney 2nd phone interview (technical)

Hey Franklin,

You are 100% confirmed for your phone interview, **tomorrow Friday, October 7th at 2pm.** You must call their conference line at **1-877-290-0784.** After that, enter this **passcode: 854-127-9651** and then your interview will begin. You will be interviewing with **Muhammad Yahia** and **Bob Rubano.** If they aren't on the a line by 2:10pm please call me so we can figure out what happened.

You are interviewing for a **Java Developer** position with **Disney** – this is a **6 to 18 month contract** position in Orlando, FL

Here is the complete job description:

6 to 18 month contract, Java Developer, Orlando, FL

General Job Description

- Looking for an individual on the DevOps team with general Java Software Engineering skills centered around build and continuous integration technologies.
- This individual must have a strong Java background but must also be able to speak multiple software development languages.
- This individual must be able to quickly troubleshoot and resolve any manner of software related issues.

Responsibilities

- Continuous Integration strategy, growth and development
- Automation Test Framework strategy, growth and development
- General Technical Leadership and Support responsibilities
- Support Strategy Development and Road-Map for technical facets
- Provide the bridge between automation initiatives and dynamic infrastructure

Skills

- 8+ years Java Development
- 6+ years experience with **JUnit** frameworks
- 5+ years with build scripting tools like **Ant** or **Maven**
- 5+ years experience with **web site/service technologies, frameworks** and **platforms**
- 2+ years experience with Continuous Integration tools like **Bamboo** and/or **Hudson**
- Proven leadership experience
- Proven troubleshooting skills and techniques
- Proven ability to quickly assess tool/technology capabilities based on historical experience
- 1+ years **Agile** SDLC experience
- Thorough understanding of **XML** structure and parsing

Things to remember:

- Be enthusiastic!!! Ask intelligent questions about the position
- Smile
- Listen attentively when asked questions
- Enunciate, and speak with conviction
- Answer questions directly and clearly
- Make sure to ask the manager "what is the next step in the interview process?" (ask this towards the end of the interview).

Things to avoid:

- Speaking too loudly or too softly
- Giving indirect answers or avoiding questions
- Asking about benefits, time off and pay

Best of luck and please call me if you should have any questions before your interview. Please send me back an email to confirm you have received all of this information and most importantly, please **call me after your interview** for an update.

Carl Dunn
Technical Recruiter
Software Resources
1325 S. International Parkway, Suite 2201
Lake Mary, FL 32746
407-515-6020 ext. 114 Office, 1-800-774-8036 Toll free
407-515-6091 Fax

We grow by referrals, so don't keep me a secret, please!!

Don't forget to go to our website and **sign up for job agents** – receive new openings daily, weekly or monthly.

Go to: www.softwareresources.com

- Select the "JOB SEARCH" tab
- Select "SIGN-UP FOR JOB AGENT" – located above the list of results


SOFTWARE
RESOURCES
CONNECTING TALENT WITH TECHNOLOGY
WWW.SOFTWARERESOURCES.COM