Memory Usage 2.0 Component Specification

1. Design

The Memory Usage component is used to obtain runtime memory usage detail for target objects. The component accepts an object and determines the total memory used. If specified, the component also determines the memory usage of each embedded object. The Memory Usage component can be used as a quick analysis tool to obtain a rough estimate of memory usage without incurring the overhead of executing a complete profiling tool.
The *Memory Usage 1.0* analyzers are hard coded to specific JVM versions. The 2.0 version adds a generic fallback analyzer, as well as new JVM specific analyzers.

The design is relatively simple. It has a main class, MemoryUsage, which exposes two methods for memory usage analysis of an object (with and without embedded objects). These methods rely on implementations of `MemoryUsageAnalyzer` interface.

The `MemoryUsageAnalyzer` interface abstracts a memory usage analyzer because the memory allocation details may differ from one JVM to another. This class will have an implementation for each supported JVM. The interface exposes a method for detecting whether the implementation can be applied to the current JVM on which the code is running. The MemoryUsage methods simply iterate each implementation until one that supports the current JVM is found.

There is also the concept of a fallback analyzer which will be used as a good approximation for any JVM (both version and vendor) The user will have the ability to specify which analyzer to use as such fallback or we could use an good average of behavior in the implementation of JDK (Sun Microsystems) 1.4 as our default fallback analyzer which we do in this design.

We have the ability to either specify the available analyzers (as well as the fallback) either through API or through configuration.

This design defines an implementation for the Sun JVMs 1.2, 1.3 , 1.4, and 1.5 as well as IBM's 1.4 and 1.5. Each has a dedicated class. While this may seem wasteful, it is easier to maintain and tweak. Since the calculations are not super exact we might at some point decide to tweak and it is easier to deal with specific class rather than methods with a number of if statements.

As you may notice in the class diagram, the base class has 7 protected methods. The reason is to provide easy extension for future JVM versions. For example, if JDK 1.5 changes the memory occupied by the fields to a more compact layout (currently a byte occupies 4 bytes, so 3 bytes are wasted), which it does. A Sun15Analyzer class extending `BaseAnalyzer` will be created. This class will need to override only the JVM detection method and some of the 4 protected getters. Everything else will remain unmodified. For example, if the array overhead changes then the `getArrayBase` method will need to be overridden to return the new value.

The `MemoryUsageResult` and `MemoryUsageDetail` class are simple structures that contain the numeric results of the analysis. `MemoryUsageResult` provides the total results (total memory usage and total object count of embedded object) while `MemoryUsageDetail` provides these results grouped for each class of objects.

The `MemoryUsageListener` class is a listener that is called during the execution of the algorithm that traverses the graph formed by the object and its embedded objects. When

a new object is reached the listener is called to see whether the embedded objects should also be processed. This provides the option to limit the depth of graph traversal.

The difficulty of this component is not the design, but the algorithm to determine the memory usage for an object. The algorithm is presented in great detail (almost Java code) in the algorithms section.

Please note that this design relies heavily in the work of the previous designer (of version 1.0) and in reality the changes were mostly architectural but the bulk of the actual design is still very much indebted to version 1.0.

1.1 Design Patterns

Strategy: since we can plug in a new implementation of `MemoryUsageAnalyzer` at any point in time without any of the `MemoryUsage` class' code changing. This can also be said of the `MemoryUsageListener`, which to some extent follows the strategy pattern as well.
Template Method: Since the `BaseAnalyzer` creates a number of protected methods, which can be overridden by a new implementation.
Listener: The `MemoryUsageListener` follows the observable pattern.

1.2 Industry Standards

None.

1.3 Required Algorithms

1.3.1 The memory usage analysis for an Object: Wrapper

The analysis algorithm relies heavily on reflection to achieve a traversal of the graph formed by the object and its embedded objects. Reflection is used to determine the non-static fields on an object and retrieve the values stored in those fields.

The traversal uses a hash set to maintain the visited objects and avoid visiting the same object multiple times. Because a hash set relies on equals to make comparisons and because the objects could override this method, an `IdentityHashMap` is used.

The example uses a breadth-first traversal using a queue in the pseudo code but the developer can use a depth-first approach (either using a stack or using recursion).

The algorithm is separated in two branches, one for arrays and one for objects.

The size of an array is the overhead `getArrayBase` (12) + the size of an element * array length. If the elements are not primitive types then they are embedded objects and they will be removed from the queue for analysis later in the process (this includes arrays of arrays (an array is an Object)).

The size of an object (non-array) is the overhead getObjectBase (8/16) + the size of each field. If a field is an Object then its value will reference an embedded object that will be removed from the queue for analysis later in the process (this included fields which are arrays).

Both the sizes of an Object and an array will be set to the nearest multiple of 8 bytes.

One issue while using reflection is access to non-public fields. Fortunately, the Field class provides the setAccessible method that will override the Java access rules while using reflection. The fields of both the class and its superclasses must be analyzed. The getSuperClass method is used to navigate up the inheritance chain to collect all the fields using the getFields method.

Of course, the pseudo code can be separated into several private methods for clarity if necessary.


```
Procedure analyze(obj, goDeep, listener) : MemoryUsageResult

result = new MemoryUsageResult
classToDetail = new HashMap
visited = {obj} (HashSet)
queue = [obj]

// process each element in queue until queue is empty
while (queue not empty){
   obj = queue.next

   // if object is an array
   if (obj.isArray){
      size = getArrayBase + getArraySize(obj) // size of the array itself

      // if elements are objects then process the elements as well
      if (goDeep && obj instanceof Object[]){
         foreach (elem in (Object[])obj){

            // if not already visited then mark it as visited and enqueue it
            if (elem != null && !visited(elem)){

               // process it only if there is no listener or
               // the listener allows it
               if (listener==null || listener.objectReached(elem)){
                  visited.add(wrapped elem)
                  queue.add(elem)
               }// if
            }// if
         }// foreach
      }//if
      // if elements are primitive then do not process the elements, simply
      // multiply the array size by the size of the primitive element.
      make size multiple of getArrayAllign
   // if object is an Object subclass
   }else{
      size = getObjectBase // size of Object + the rest
      fields = new Set

      // the trick here is to walk all the superclasses of the class
      // and include their fields also
      // (because Class.getFields returns only the public inherited fields
      // we have to rely on getDeclaredFields which returns all the fields
      // and implement ourselves the retrieval of the inherited fields)
      for (cls = obj.getClass(); cls != null; cls = cls.getSuperClass){

         // we don't care about static fields
         foreach (field in cls.getDeclaredFields and field non-static){
            fields.add(field)

            // if it's not primitive then we need to process the embedded
            // object as well
            if (goDeep && !field.isPrimitive){

               // another trick - allows us to access fields regardless
               // of their visibility (even private !!!)
               modifiedVisibility = field.getAccessible()
               if(modifiedVisibility == false){
                  field.setAccessible(true)
               }//if

               // get the value and if it wasn't already visited then
               // mark it as visited and enqueue it
               value = field.getValue(obj)
               if (value != null && !visited(value)){

                  // process it only if there is no listener or
                  // the listener allows it
                  if (listener==null || listener.objectReached(elem)){
```

```
                        visited.add(wrapped value)
                        queue.add(value)
                     }//if
                  }// if
                  // we need to remember to modify the visibility of the field
                  // back to what it was (of private) with field.setAccessible(false)
                  if(modifiedVisibility == true){
                     field.setAccessible(false)
                  }
               }//if
            }//foreach

            Sort(fields)
            foreach (field in fields) {
               size = align(size, getFieldSize(field))
               size += getFieldSize(field)
            }
         }//for
         make size multiple of getObjectAllign
      }//else

      // cumulate object size to the result
      result.cummulate(1, size)

      // cumulate object size to the detail for its class
      detail = classToDetail.get(obj.getClass)
      if (detail == null){
         detail = new MemoryUsageDetail(obj.getClass)
         classToDetail.put(obj.getClass, detail)
      }//if
      detail.cummulate(1, size)

   }//while

   // return results
   result.setDetails(classToDetail)
   return result
```

The getFieldSize method will return the following values:

```
For boolean, byte, char, short, int, float - 4 bytes
For long, double - 8 bytes
else (Object, arrays) - 4 bytes
```

The getArraySize method will return the following values:

```
boolean[] - 1 * length bytes
byte[]    - 1 * length bytes
char[]    - 2 * length bytes
short[]   - 2 * length bytes
int[]     - 4 * length bytes
float[]   - 4 * length bytes
long[]    - 8 * length bytes
double[]  - 8 * length bytes
else (Object[], array of array) - 4 * length bytes
```

The algorithm needs to be slightly modified for JVM 1.4 because it performs a sort on the sizes of the fields and groups primitives with sizes less than 4 bytes together, achieving this way both 4-byte alignment for 4-bytes size or more types and not wasting so much space. (This is also true for 1.5) For example two char fields will occupy 4 bytes in JVM 1.4 and 8 bytes in JVM 1.3.

1.3.2 Getting the objects embedded in an object

The MemoryUsage class provides a method named getEmbeddedObjects to allow the user to obtain the list of the objects referenced by a given object. The previous algorithm

includes all the techniques needed to achieve this.

Essentially the values of all the fields of the given object must be obtained (using the method from the previous algorithm) and added to a HashSet (objects have to wrapped in a dummy class as in the previous algorithm). Finally, an array built from the set (unwrapping the objects from the container class) is returned.

1.4 Component Class Overview

1.4.1 com.topcoder.util.memoryusage

Class MemoryUsage:

The memory usage class provides static methods to obtain the memory usage for an object (including or not including the embedded objects recursively). This class determines the appropriate analyzer implementation for the running JVM. The selected analyzer will be called to perform the actual memory usage analysis.

Interface MemoryUsageAnalyzer:

This is an Interface that abstracts an analyzer for the memory usage of an object. It exposes a method to detect whether the implementation applies to the running JVM and two methods for analyzing the memory usage of an object (including or not embedded objects recursively).

Class BaseAnalyzer:

Base Memory usage analyzer implementation, which creates a common base for most analyzer, related activity. This is basically an abstracted out Memory usage analyzer implementation. All analyzers will extend this class which gives the convenience of having the traversal algorithm already implemented for example.

Class MemoryUsageResult:

This class provides the results for the memory usage of an object. It exposes getters for the used memory in bytes and the object count. It also provides the option to obtain memory usage details for objects of a particular class.

Class MemoryUsageDetail:

This class provides detailed memory usage for the instances of a specific class (including primitive types and arrays). Provides methods for getting the class represented by the memory usage detail, the number of instances of that class and the memory occupied by all the instances together.

Class MemoryUsageListener:

Listener for events generated during the object graph traversal. When an object is reached then the objectReached event is called to signal that the object should be processed or the traversal should not go into its embedded objects. The purpose is to set a customizable limitation for the graph traversal (especially when the graph is very large).

1.4.2 com.topcoder.util.memoryusage.analyzers

Class Sun12Analyzer

Memory usage analyzer implementation for the Sun JVMs 1.2.
There are no differences between the 1.2 memory model and the one in the BaseAnalyzer so we only override/implement the matchesJVM method.
Thread-safe since it has no state.

Class Sun13Analyzer

Memory usage analyzer implementation for the Sun JVMs 1.3. There are no differences between the 1.3 memory model and the one in the BaseAnalyzer so we only override/implement the matchesJVM method.
Thread-safe since it has no state.

Class Sun14Analyzer

Memory usage analyzer implementation for the Sun JVMs 1.4. There are only slight differences between the 1.4 memory model and the one in the BaseAnalyzer so we only override/implement the following methods:

1. matchesJVM method.
2. getFieldSize() since in jdk 1.4 field size is the actual size (it is not padded to a word - 4 byte length)

Thread-safe since it has no state.

Class Sun15Analyzer

Memory usage analyzer implementation for the Sun JVM 1.5.x There are only slight differences between the 1.5 memory model and the one in the BaseAnalyzer so we only override/implement the following methods:

1. matchesJVM method.
2. getFieldSize() since in jdk 1.4 field size is the actual size (it is not padded to a word - 4 byte length)

Thread-safe since it has no state.

Class IBMn14Analyzer

Memory usage analyzer implementation for the IBM JVM 1.4.x There are only slight differences between the IBM 1.4 memory model and the one in the BaseAnalyzer so we only override/implement the following methods:

1. matchesJVM method.
2. getFieldSize() since in jdk 1.4 field size is the actual size (it is not padded to a word - 4 byte length)

Thread-safe since it has no state.

Class IBM15Analyzer

Memory usage analyzer implementation for the IBM JVM 1.5.x. There are only slight differences between the 1.5 memory model and the one in the BaseAnalyzer so we only override/implement the following methods:

1. matchesJVM method.
2. getFieldSize() since in jdk 1.4 field size is the actual size (it is not padded to a word - 4 byte length)
3. getObjectBase() since the object header is 16 bytes rather than the previous 8

Thread-safe since it has no state.

1.5 Component Exception Definitions

Exception MemoryUsageException:

Exception thrown by the implementations of the memory analysis algorithms. The algorithms rely heavily on reflection that can generate exceptions related to security issues while accessing field values if the access policies are modified. Propagating each exception exposes the user to too much implementation details. As such, this class serves as a wrapper for those exceptions. If the user wants to get into details then the wrapped exception can be obtained.

- MemoryUsage.getShallowMemoryUsage method - propagated from the implementation of the analyzer (generally wraps an security related exception)
- MemoryUsage.getDeepMemoryUsage method - propagated from the implementation of the analyzer (generally wraps an security related exception)
- MemoryUsageAnalyzer.getShallowMemoryUsage method - wraps a reflection exception (generally security related)
- MemoryUsageAnalyzer.getDeepMemoryUsage method - wraps a reflection exception (generally security related)
- BaseAnalyzer.getShallowMemoryUsage method - wraps a reflection exception (generally security related)
- BaseAnalyzer.getDeepMemoryUsage method - wraps a reflection exception (generally security related)
- BaseAnalyzer.analyze - wraps a reflection exception (generally security related)

Exception JVMNotSupportedException:

Exception thrown to signal there is no analyzer for the running JVM.

- MemoryUsage.getShallowMemoryUsage method - if no analyzer supports the running JVM
- MemoryUsage.getDeepMemoryUsage method - if no analyzer supports the running JVM

Exception ConfigurationException

This exception is thrown when there are issues with provided configuration.

- There was an issue with configuration provider itself (i.e. `ConfigManager` threw an exception)
- Required parameters were not supplied (like a slit of analyzers)

Exception IllegalArgumentException:

This exception is thrown mostly when null arguments are encountered.

1.6 Thread Safety

The component is thread-safe in the sense that there is no state held during computations.. All the methods are help to be almost like utility methods. Thus in the MemoryUsage (main class) all the methods are Getxxx.

Other classes are mutable but they mostly are used internally or as a cumulative data structure for a single thread (intended usage).

The object being analyzed should not change while the analysis is being done on it. In other words it is expected that the user will not introduce side effects. Currently the implementation never locks on any objects.

2. Environment Requirements

2.1 TopCoder Software Components:

Config Manager 2.1.4: Used for configuration of the analyzers as well as fallback analyzer.

Object Factory 2.0: Used to create actual MemoryUsageAnalyzer implementations.

BaseException 1.0: Use to create custom base for all thrown exceptions that can be chained or wrapped.

2.2 Third Party Components:

- None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.memoryusage

3.2 Configuration Parameters

Parameter	Description	Values
analyzers	List of analyzer Object Factory tokens **required.**	Keys used by Object Factory to instantiate the analyzers
analyzers_namespace	Object Factory namespace. **optional.**	Any valid namespace. Defaults to the namespace used in construction of MemoryUsage + `".objectFactory"`
fallback_analyzer	Object Factory instantiation token for a single analyzer **optional.**	A key to instantiate an analyzer
default_fallback_analyzer_flag	flag stating if a default fallback analyzer should be used (true) or not (false) **optional.**	true/false

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

The usage is very simple. The class MemoryUsage exposes 2 static methods for determining the memory usage of an object. The method getShallowMemoryUsage doesn't include embedded objects and the method getDeepMemoryUsage includes embedded objects. These methods return an MemoryUsageResult object which provides

the methods getUsedMemory() and getObjectCount() to retrieve the used memory and the object count. Details about the embedded instances grouped by classes can be obtained using getDetails or getDetail(Class).

4.3 Demo

Assume the following simple configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<CMConfig . . .>
   <Config name="some.namespace.com">
         <Property name="analyzers">
                  <Value>Sun12Analyzer</Value>
                  <Value>Sun13Analyzer</Value>
                  <Value>Sun14Analyzer</Value>
                  <Value>Sun15Analyzer</Value>
                  <Value>IBM14Analyzer</Value>
                  <Value>IBM15Analyzer</Value>
         </Property>
         <Property name="analyzers_namespace">
                  <Value>some object factory namespace</Value>
         </Property>
         <Property name="fallback_analyzer">
                  <Value>Sun14Analyzer</Value>
         </Property>
         <Property name="default_fallback_analyzer_flag">
                  <Value>false</Value>
         </Property>
   </Config>
</CMConfig>
```

4.3.1 *Creating and configuring*

```
//
// We can create the memory usage based on configuration:

// default namespace
MemoryUsage memoryUsage = new MemoryUsage();
// specific namespace
MemoryUsage memoryUsage = new MemoryUsage(“some.namespace.com”);
// API based configuration, use default fallback
MemoryUsage MemoryUsage = new MemoryUsage(list of analyzers, null, true);
// API based configuration, specific fallback
MemoryUsage MemoryUsage = new MemoryUsage(list of analyzers, new
Sun12Analyzer(), false);
```

4.3.2 *Computing the memory usage*

```
// analyze memory usage
MemoryUsageResult r = memoryUsage.getShallowMemoryUsage(object);
     or
MemoryUsageResult r = memoryUsage.getDeepMemoryUsage(object);

// print totals
System.out.println(“used memory: “ + r.getUsedMemory());
System.out.println(“object count: “ + r.getObjectCount());

// get into details
MemoryUsageDetail[] d = r.getDetails();
for (int i = 0; i < d.length; i++) {
    System.out.println(“class: “ + d[i].getDetailClass().getName()
```

```
                    + " instances: " + d[i].getObjectCount()
                    + " used memory: " + d[i].getUsedMemory());
        }
```

4.3.3 *Listener utilization*

```
// Assume a simple listener prints out encountered class name
MemoryUsageListener myListener = new MemoryUsageListener(){
  public boolean objectReached(Object obj){
          System.out.println(obj.getClass().getName());
}}

// Use it
MemoryUsageResult r = memoryUsage.getDeepMemoryUsage(object, myListener);
```

5. Future Enhancements

- Add more accurate estimations
- Add more platforms.