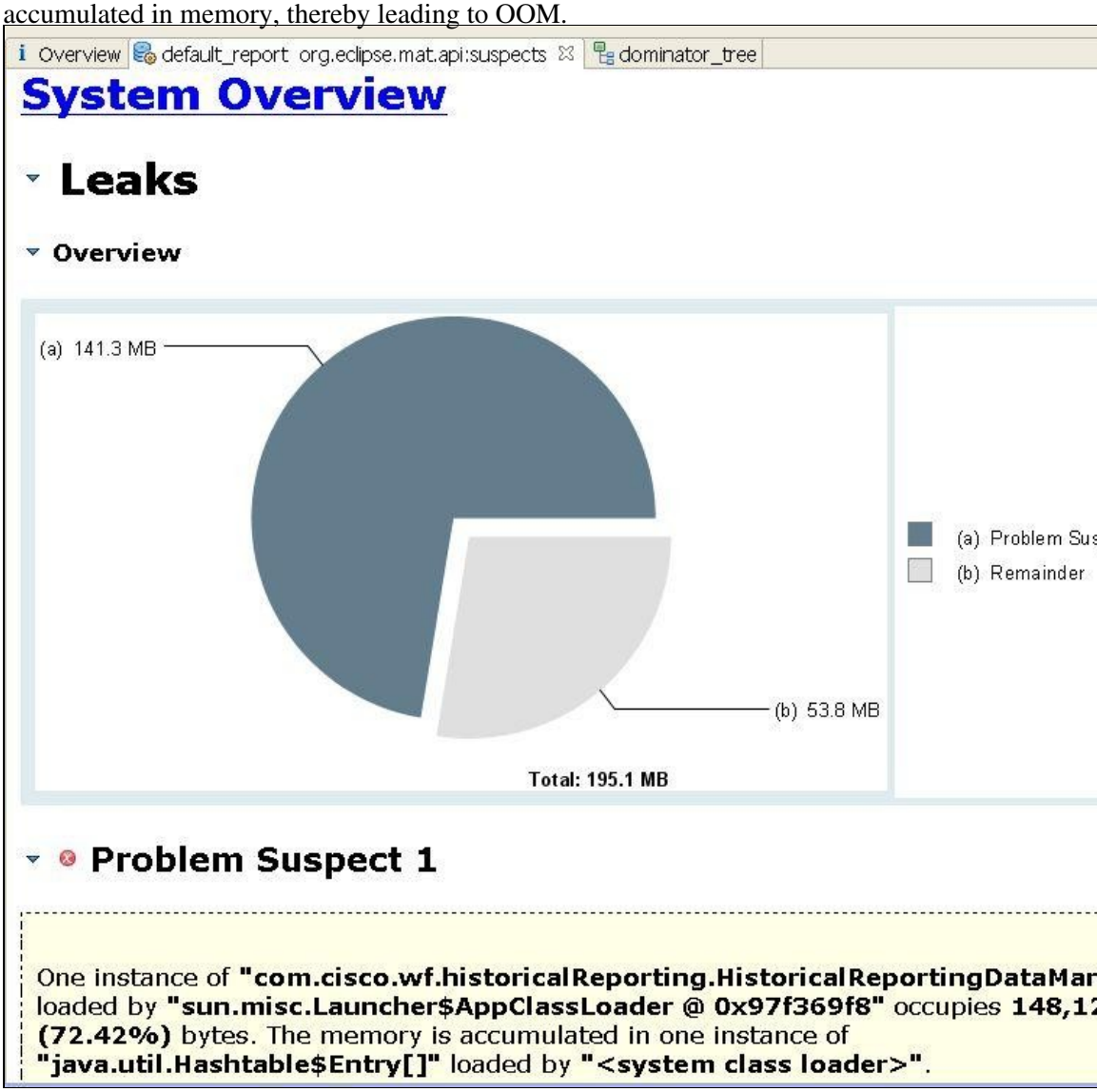
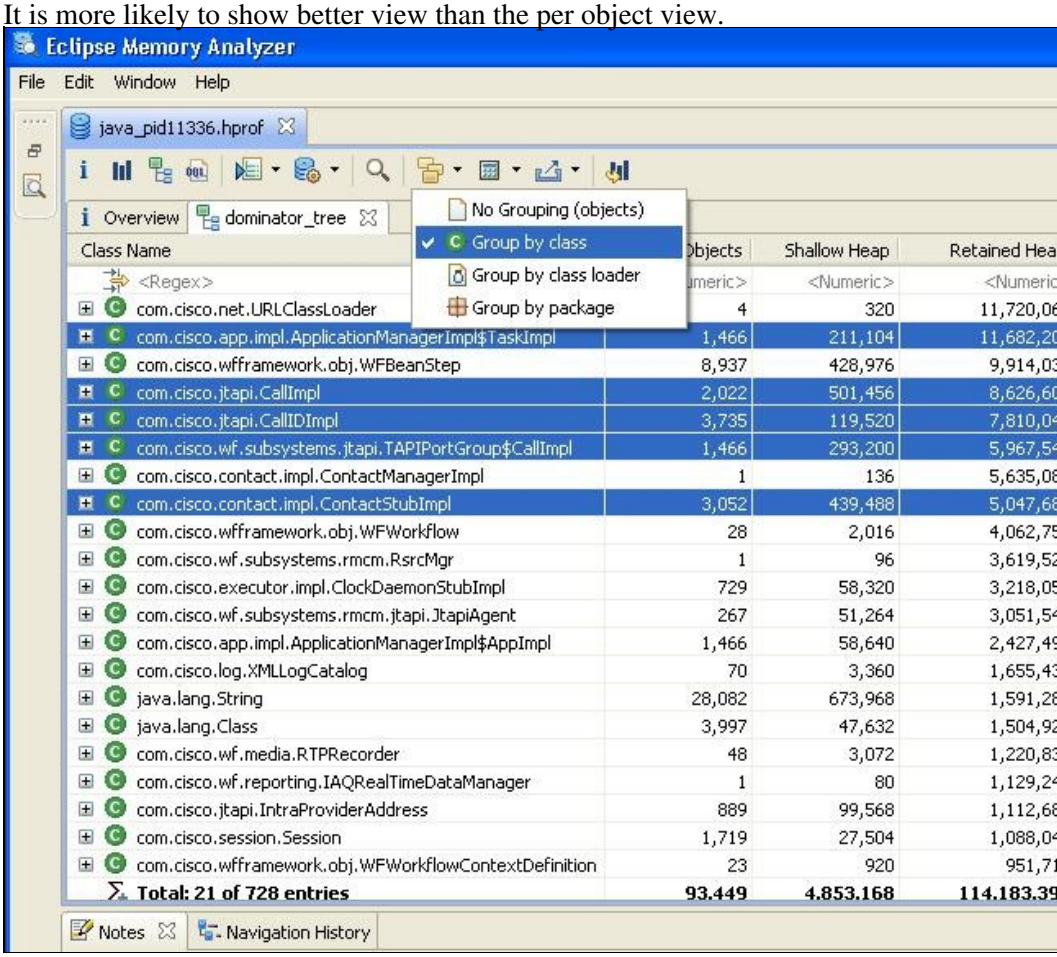
Eclipse Memory Analyzer is a very good tool for analyzing the heap dumps (.hprof files). Following is a guideline for the initial analysis of the heap dump.

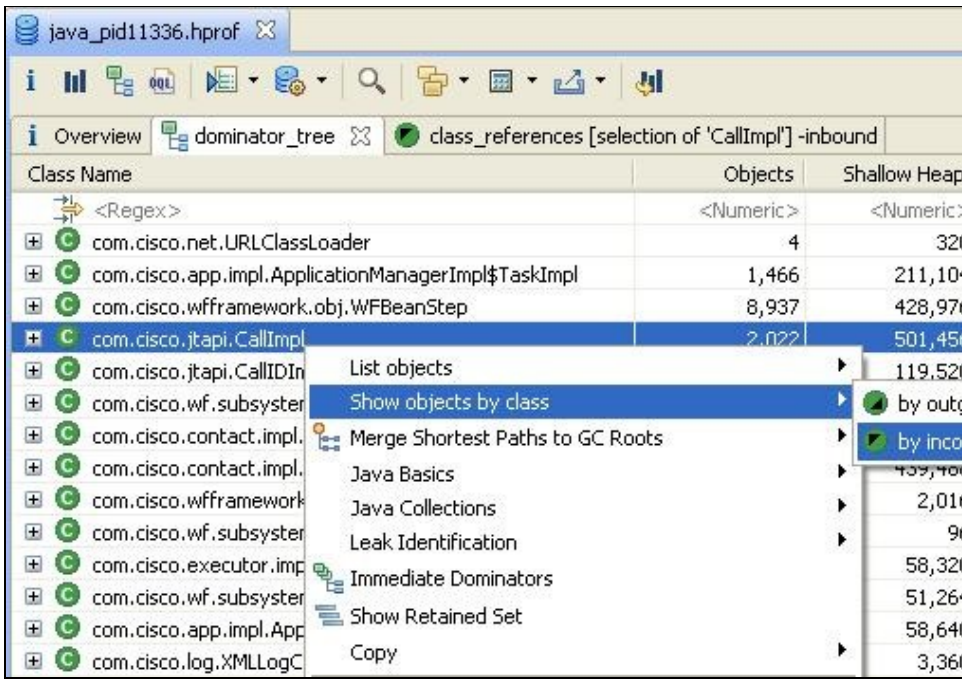
- How to get the Eclipse Memory Analyzer Tool (MAT)?
  - ♦ Download it from http://www.eclipse.org/mat/ and install it on your desktop.
- How to open the dump?
  - ♦ Start Memory Analyzer and select ?Open Heap Dump? from the File menu.
    - ◊ MAT parses the heap dump and opens a few default reports.
    - ◊ The pie in the overview pane shows the distribution of retained memory on a per-object basis. It shows the biggest objects in memory (objects that have high Retained memory - memory accumulated by it and the objects that it references).
- Leak Suspects? - **Find the memory leak**
  - ♦ If the leak is induced by a single component, it would show up in the pie graph in the overview pane itself.
    - ◊ In the following instance, historical call data was not written to DB and they were accumulated in memory, thereby leading to OOM.
    - ◊ 

    - ◊ It is always easy to find and fix such problems :-)
  - ♦ What if the leak is not induced by a single/few objects
    - ◊ Leak suspect may not show up in the overview reports, or in the top of the dominator tree.
    - ◊ The leak could be induced by a class of objects (i.e. leak induced per call).

· Open the dominator tree from the toolbar using the button  .
· Select "Group result by..." from the toolbar and select "Group by class".
· It is more likely to show better view than the per object view.
·

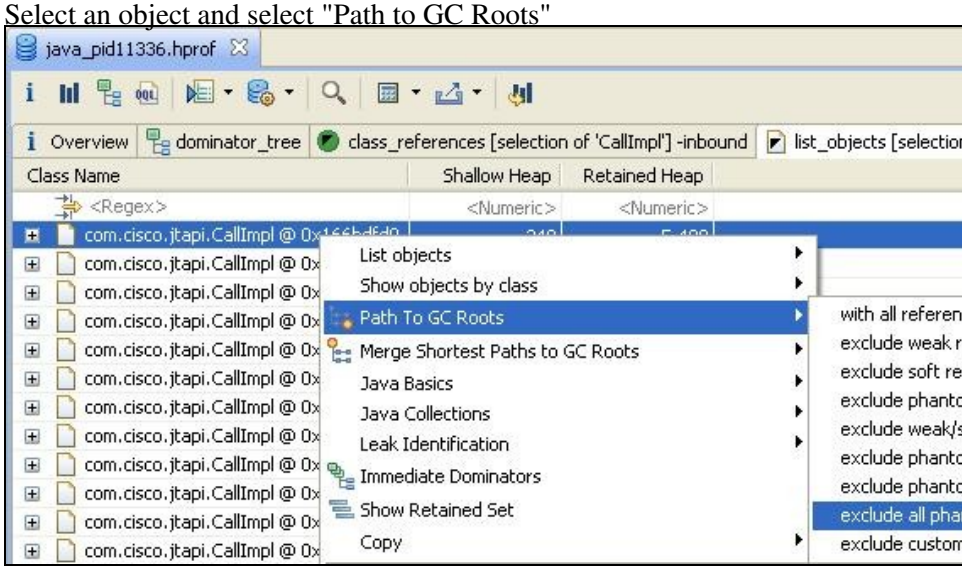


· Once a suspect class is found, list its objects and find the objects holding reference to it.
    - Select List objects ?> with incoming references from the context menu. As a result we will get an object reference graph.
    - Drill down the graph to find the references.
    - If we are analyzing the references of a single class, it is better to use "Show objects by class -> with incoming references".

- 

  - Use "Immediate Dominators" to get the list of objects referencing it. You can exclude the packages that you are not interested in.

· The easiest way to find out what keeps these leaking objects alive is to check the path to the garbage collection roots.

  - Select an object and select "Path to GC Roots"
  - 

  - To find it at the class level, use the "Merge Shortest Path to GC Roots" query. The Merged Paths to GC Roots view shows the shortest paths from the GC roots to each instance of the selected class.
    - ♦ The thread that is holding the reference is shown. Drill down to find the top level object referencing our suspect class, and drill down further to find our suspect class and its immediate references.
  - Both the per object and class level path to GC roots gives us an idea of bottom-up and top-down references to the suspect class.

♦ **Use OQL for any custom queries**.

◊ In one of the customer setup, most of the heap was occupied by Workflow steps. It could be either due to uploading too many scripts, or due to excessive usage of steps per workflow.

- Following is an OQL query to list the scripts uploaded, no. of steps per script, and the memory occupied by each script.
  - *SELECT toString(wf.script.name) AS Workflow_Script_Name, wf.steps.nSize AS Workflow_Steps_Count, wf.@retainedHeapSize AS Workflow_Mem_Occupied FROM com.cisco.wfframework.obj.WFWorkflow wf*
- Following is an OQL query to find which scripts did the customer configure as sub-flows, and their corresponding main script.
  - *SELECT toString(sf.subflowExpr.text) AS SubFlow_Name, toString(sf.container.script.name) AS MainFlow_Name FROM com.cisco.wfframework.steps.core.StepCallSubflow sf*
- Similar queries can be devised by going through the object graph and using the Inspector window to look at the values of the object attributes.

♦ **Check for Finalizers**

◊ Finalizable objects are reclaimed only after a few GC cycles since when the object is discovered as unreachable.

- If GC discovers that a finalizable object is unreachable, it adds it to finalization queue, from which the finalizer thread retrieves the objects to finalize. The object gets reclaimed only during the GC later to the finalizer thread marking the object as finalized.

◊ Ideally a component should implement finalize() only when it is really required. Also it should be small and should not reference large objects, otherwise the referenced objects too will not get GCed until this object is finalized.

◊ Select "Java Basics -> References -> Finalizer Reference Statistics" to get an overview of which objects implemented finalization, their retained heap, objects ready for finalization, etc.

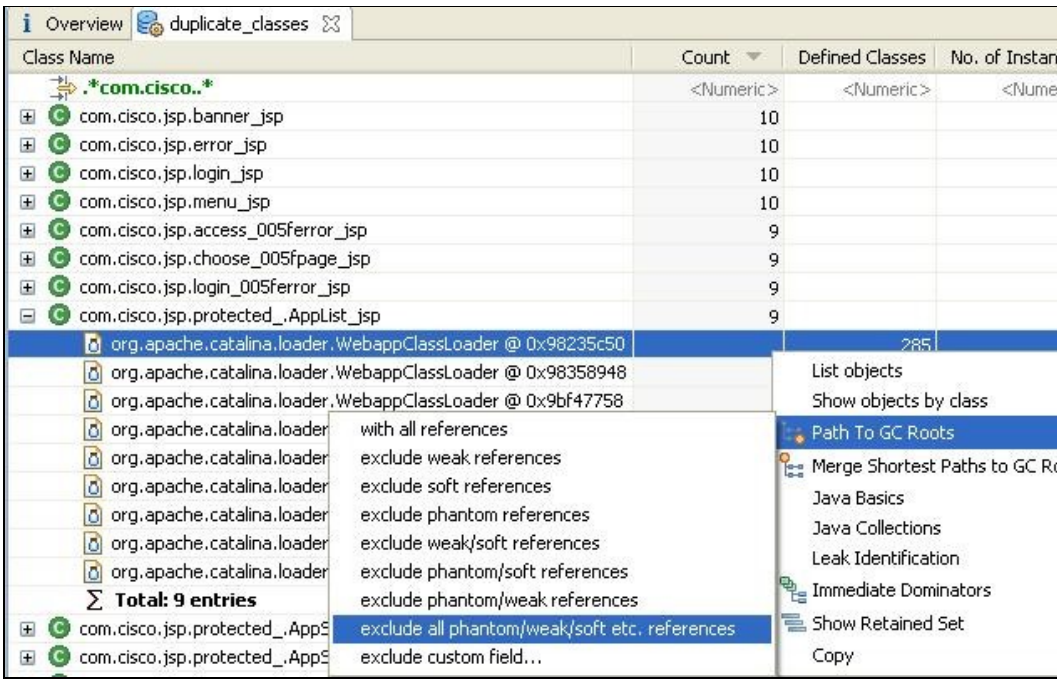
♦ **PermGen issues - ClassLoader leak**

◊ Container like Tomcat uses a separate Classloader for loading each webapp. If this classloader does not get GCed when the webapp is stopped/restarted, it is a classloader leak.

◊ Select "Java Basics -> Class Loader Explorer" to get an overview of the list of classloaders, the classes that they defined, and the no. of objects loaded.

- To check the references to this Classloader, right click on the classloader and select "ClassLoader -> Path to GC Roots".
- If the references are only held by a thread's contextClassLoader, it could be due to the improper setting of the thread's contextClassLoader.

◊ Select "Java Basics -> Duplicate Classes" to get a list of classes that were loaded by multiple Classloaders.



- If you notice that a class should have been loaded only once, you can identify the leaking Classloaders by checking the "Path to GC Roots" of the Classloaders that loaded the duplicate class.

- **Performance issues**
  - Select "Java Basics -> Thread Overview" to get the list of the threads that were running at the time of dump generation, and the objects that they are referencing.
  - Use "Java Collection -> *" to get a report on what is the size of each collection, fill ratio of each collection, collision ratio of the maps, objects referencing each collection, etc.
    - This will give an idea on whether the collection used is an appropriate one, whether it's initial size is properly set, whether the objects implemented proper hashcode, etc.