Eclipse Visualization and Performance Monitoring

Chris Laffra
IBM Ottawa Labs

http://eclipsefaq.org/chris
Roadmap

- Introduction
- Introspection
- Java Spider
- Visualization
- Profiling
- Conclusions
Introduction

- Eclipse is a massive framework
- Learning curve may be steep
- How to make sense from 100s of plug-ins
- Key to understanding performance are insights into operating systems, Java language, and all constructs above
- Management of limited resources
Eclipse Concepts

- Plugins Dir
- Plugin Subdir
- Manifest File
- Library
- .Zip
- Directory
- .Jar
- Plugin
  - name
  - vendor
  - id
  - version
  - class
- dependants
- requires
- describes
- runtime
- Extension-Point
  - id
  - name
  - markup XML
  - interface
- Extension
  - point
  - id
  - markup XML
- 0..n
- 1..1
- extends
Introspection

To find out what happens in Eclipse, use:

- The platform core tools:
  - www.developerworks.ibm.com search for ‘Kehn’
- A debugger and step into and over methods
- Insert System.out.println() calls
- A Java profiler based on JVMPI
  - JProbe/Yourkit/Hyades/XRay, etc.
- A tool/classloader that instruments bytecodes
  - AOSD: weave in a concern
  - Eclipse monitor based on JikesBT (IBM alphaworks)
Inspect Eclipse: Runtime Spy

Part of Eclipse core Tools
Shows Plug-in Load order
Inspect Eclipse: Java Spider

- Written by Erich Gamma and Kent Beck
- [http://www.javaspider.org](http://www.javaspider.org)
Profiling: JVMPi and JVMTI

- JVMPi – Java VM Profiling Interface
  - VM runs slower – less aggressive JIT
  - Must be able to generate events on demand
  - JVMPi agent can turn on/off any event at will
  - Makes VM design less clean

- JVMTI – Java VM Testing Interface
  - Uses bytecode instrumentation done by VM
  - Choose ahead of time what you want to trace
  - Clean VM design
  - May completely confuse JIT
The Cost of Profiling

- All profiling processes are intrusive
- JVMPi reduces JIT and may completely change the behavior of the target
- Instrumentation can be equally distorting
- Sampling technique are less intrusive, but provide only hot spots and have a hard time to provide call hierarchies, *etc.*
- Summary: use profiler output only as a hint to start looking at trouble spots
Profiling Eclipse: Products

- OptimizeIt
- Quantify
- JProbe
- Yourkit
Profiling Eclipse: Open Source

Eclipse Profiler (SourceForge)

Hyades (Eclipse project)

EJP (SourceForge)
Profiling Eclipse: XRay

- Modeled after “Task Manager”
- Low Overhead – Enabling Daily Use
- Show Indicators in one Single Graph, Allow Human Brain to Observe Patterns
- Generate Reports with Screenshots
XRay: Vital Statistics

- **CPU% of process**
- **Heap Use in MB**
- **Disk Read and Write**
- **Windows Handles**
- **Burn Rate in MB/s**

**Combined Graph**
- Showing CPU, burn rate, Disk I/O, Plugin Load, GC, and Total Live Objects

**Thread View**
- Showing Activity Per Thread If It did Something in the second shown

**Report Area**
- Showing Details when a time slot is selected in main Graph

---

The green line indicates JVM CPU usage (0-100%). The white line indicates object allocation burn rate (0-10MB/s). The cyan line indicates disk I/O (0-10MB/s). The dark yellow line indicates total of allocated objects. The red markers indicate GC occurrence. The yellow markers indicate plugin activation. Click the graph (or the Report tab) to see more details. The indicators at the left show current vitals.
XRay: Plug-in Profiling

1. Browse top-down or bottom-up
2. Move focus from Plug-ins to methods
3. Find “Hot” Plug-ins
4. Find “Hot” Callers
XRay: Plug-in Profiling

- Data sorted on a plug-in level so a user can more easily identify their code
- Two modes of drilling down:
  - Find out for a given plug-in who is calling you
  - Find out all the plug-ins called by your plug-in
- Search methods by name and top 1000
- Data: CPU time, #calls, #news, #classloads, bytes read, bytes written
**XRay: Comparing 2 Runs**

- **Visual differentiation cues**
- **Filtration mechanism**
- **Concrete relative figures**

**Table Comparison**

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Value</th>
<th>Difference</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile time (ms)</td>
<td>3.15600</td>
<td>3.15600</td>
<td>0.00</td>
</tr>
<tr>
<td>Objects Allocated</td>
<td>2794</td>
<td>2794</td>
<td>0.00</td>
</tr>
<tr>
<td>Classes Loaded</td>
<td>6</td>
<td>6</td>
<td>0.00</td>
</tr>
<tr>
<td>Bytes Read</td>
<td>3223011</td>
<td>3223011</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Performance Report**

- Profiling
- Testing
- Error Log
- ProfileFolder
- Comparison
XRay: Finding Methods

- Enter ‘ActionSet’
- Finds all methods and sorts by CPU
- Clicking on a result to show method in Profile Tree and Method Details.
**XRay Example:**

1. Load empty workbench
2. Switch to Java Perspective (first time)
3. Create a new Project (first project). This triggers Java indexing and types caching (reads rt.jar)

---

Event selection from 10:55:41 to 10:56:22 (which is 40 seconds). During this time:

- 7 plugins were loaded: org.eclipse.team.core, org.eclipse.jdt.launching, org.eclipse.team.cvs.
- 772 classes were loaded.
- 28,877,808 bytes were read from disk in 34934 operations.
- 6,976,520 bytes were written to disk in 13743 operations.
- 6,454,471 new Java objects were created, needing 440,123 Kbytes on the Java heap.
- 30,294 CPU msecs were spent, of which 6,340 CPU msecs (20%) inside GC.
- 323 methods were compiled by the JIT, for a total of 1,949,865 bytes of native machine code.

Click on the 'Report' tab above to see more details...

The green line indicates JVM CPU usage (0-100%).
The white line indicates object allocation burn rate (0-10MB/s).
Visualization

- Experience has shown that best technique is to use bytecode instrumentation
- Approach is special classloader that takes bytecodes and inserts extra code
  - Could use BCEL, JikesBT, or other tools
- Need to cache the result to improve startup time for future runs
- May need a process to instrument all Eclipse
JikesBT – Quick Overview

- Bytecode represented as a graph
- Makes it very easy to add calls anywhere
- Expensive process, explodes class file into complex Java structure with many Strings
Interesting Events?

- Entering a method
- Value of method parameters
- Creation of new objects
- Call to a method which is known to be outside of instrumented code
- Leaving a method (with return value)
Applet Dashboard Example

- Hijack the IE AppletClassLoader
- Add ability to suspend, kill, trace, and rehost them
- Weave in concern to detect network access
Domain-specific Visualization

- Take a ‘model’ and add a new ‘view’ by adding a new concern
- Good: Does not complicate original code
- Allows for specialized representations and control
Howto: Eclipse Visualization

Adding visualization to Eclipse:

- Add a new OSGi adaptor to Eclipse
- Overload the ‘defineClass’ method:
  - Obtain instrumented code from cache, or
  - Use bytecode instrumentation
  - Call super.defineClass on the result
- Implement runtime loaded by bootloader
- Connect visualizer to runtime
Eclipse Visualization

- Showing Plug-ins when they are loaded
- Indicate activity and interaction between plug-ins
- Light-weight, little overhead
- Allows for drilling down into specific plug-ins
Eclipse Visualization

1. Filter plug-ins to show methods traces
2. See time stamps
3. Method names
4. Argument values
5. Call to other plug-ins
6. Object creation
7. Return values
Eclipse Visualization

- Inspect the entire Java heap
- See what plug-in objects are created and when
- Inspect values
- Find memory leaks
## Eclipse Visualization

- Simple extension point and wizard allow for custom visualization.
- 2 samples: Profiler and Eclipse Disco
Performance Tips

- Use the right tools:
  - Profiler, instrumentation, or System.out
- Be Lazy: Avoid “touching” plug-ins, classes and static initializers (avoid class loading)
- Avoid doing things until you really have to
- Some things are much slower than others
- Don’t go overboard (use 90/10 rule & focus)
- Most importantly: Don’t over-generalize!!!
Slow Things in Eclipse/Java

- Bad Algorithms. Linear Algorithms.
- Deep recursion (due to Java stack)
- Things like Hashtable/Vector (synchronized). Use HashMap/ArrayList instead
- Class Loading and Initialization
- Lots of object creation/garbage collection
- I/O (buffer your reading/writing). Always test on a ‘cold’ disk.
Slow Things in Eclipse/Java

- Method calls (compared to Field access)
- Flexible, object-oriented architectures (do you really need that design pattern here?)
- JNI calls (calling them, not running them)
- Complex UIs (avoid multiple refreshes) – GEF is known for deeply recursive layout
- Too many threads
Fast Things in Eclipse/Java

- Local variables, fields, loops, arrays
- Exception clauses (try-catch)
- Small code
- Running JNI calls or native (AOT) code
- Native widgets (think SWT vs. Swing)
- Anything that avoids class loading (plug-in activation, and disk access)
- Fastest code is code that is not executed
Final Wise Words

“Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.”

-- Edsger Dijkstra
References

- http://eclipse.org
- http://eclipsefaq.org