Continuous Performance –
Monitoring Performance with Automated Tests

André Weinand
Christof Marti
IBM Research
Zurich, Switzerland
{andre_weinand,christof_marti}@ch.ibm.com

Sonia Dimitrov
IBM Rational Software
Ottawa, Canada
sonia_dimitrov@ca.ibm.com
Motivation

- Performance comes always late in release cycle
  - Last minute performance “scrunches”
  - Trading performance against stability, memory footprint, etc.

- Eclipse 3.1 goal:
  Continuous performance monitoring right from the beginning of the release cycle
  - Small and simple infrastructure
  - “Red/green indicator”
  - Historical data: “When did we get slower?”
  - Build upon what developers know: JUnit

- New plug-in: org.eclipse.test.performance
Overview

- Architecture
- How to write performance tests?
- Running tests and collecting data
- Presenting performance results
- Interpreting performance results
- Pitfalls
Architecture Overview

Eclipse

- Performance Tests
  - JUnit
  - Performance Plug-In
    - JDBC
    - Remote Driver

Derby Server
  - Performance Data

Release Engineering Visualization Tools
Prepare for writing a performance test

- Check out from dev.eclipse.org:/home/eclipse
  - org.eclipse.test.performance
  - org.eclipse.test.performance.win32 (on Windows)

- Get org.junit plug-in (e.g. from Eclipse install)

- Add `org.eclipse.test.performance` and `org.junit` to your test plug-in’s dependencies
Writing a performance test

```java
public void testMyOperation() {
    Performance perf = Performance.getDefault();
    String ID = perf.getDefaultScenarioId(this);
    PerformanceMeter meter = perf.createPerformanceMeter(ID);
    try {
        for (int i = 0; i < 10; i++) {
            meter.start();
            // code to measure
            meter.stop();
        }
        meter.commit();
        perf/assertPerformance(meter);
    } finally {
        meter.dispose();
    }
}
```

Asserts that collected data is within [-100%, +10%] range of reference data in the performance database.
Writing a PerformanceTestCase

- Convenience class **PerformanceTestCase**

```java
public class MyPerformanceTestCase extends PerformanceTestCase {

    public void testMyOperation() {
        for (int i = 0; i < 10; i++) {
            startMeasuring();
            // code to measure
            stopMeasuring();
        }
        commitMeasurements();
        assertPerformance();
    }
}
```
Performance Meters

- A Performance Meter
  - Monitors performance counters across a single piece of code
    - Piece of code identified by “scenario name”
  - Performance counters are called “dimensions”
  - Default dimensions provided by OS:
    - Elapsed time, kernel/user time
    - Memory consumption, VM characteristics
    - I/O (bytes read, written)
  - Multiple calls to `start()`/`stop()` are averaged
  - `commit()` prints collected data:

<table>
<thead>
<tr>
<th>Scenario 'A' (average over 10 samples):</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Time: 6 ms</td>
</tr>
<tr>
<td>Used Java Heap: 2K</td>
</tr>
<tr>
<td>Kernel time: 1 ms</td>
</tr>
</tbody>
</table>
Performance Meters (contd.)

- Don’t reuse PerformanceMeters for more than one piece of code
  - Create a new instance!
  - Distinguish via scenario name
  - Call `commit()` on all meters outside measured code, after last call to `stop()`
Performance Summaries

- Tag PerformanceMeter for global summary:

```java
// ...
_performanceMeter meter = _performance.createPerformanceMeter(_ID);
_performance.tagAsGlobalSummary(meter, "A Short Name", _Dimension.CPU_TIME);
//...```

- Tag PerformanceMeter for component summary:

```java
// ...
_performanceMeter meter = _performance.createPerformanceMeter(_ID);
_performance.tagAsSummary(meter, "A Short Name", _Dimension.CPU_TIME);
//...```
Running Performance Tests

- As part of Eclipse Automated Testing Framework on each build
  - Add to Ant target “performance” to test.xml:

  ```
  <!-- performance test target -->
  <target name="performance">
  ...
  <ant target="ui-test" antfile="${library-file}" dir="${eclipse-home}"/>
  <property name="data-dir" value="${workspace-dir}"/>
  <property name="plugin-name" value="${test-plugin-name}"/>
  <property name="classname" value="${test-case}"/>
  </ant>
  ...
  </target>
  
  • Performance data is written to database on PerformanceMeter#commit()

- Locally
  - Create Launch Configuration
  - Specify -Xms256M -Xmx256M to avoid memory pressure during measurements
  - Performance data is written to console on PerformanceMeter#commit()
Collecting data in database

- Create “Derby” library project (for details see: Performance How-To document)

- Configure performance plug-in by setting three system properties:
  - Database location:
    - `-Declipse.perf.dbloc=<location of server>`
  - Store tagged data as “Variations”
    - `-Declipse.perf.config=<key1>=<value1>;<key2>=<value2>;...;<keyn>=<valuen>`
  - Assert performance against reference data
    - `-Declipse.perf.assertAgainst=<key1>=<value1>;<key2>=<value2>;...;<keyn>=<valuen>`

- Example:
  - Store reference data for 3.0 build in DB
    - `-Declipse.perf.config=platform=win32;build=N20040625;jvm=sun`
  - Store new data and compare it against reference data
    - `-Declipse.perf.config=platform=win32;build=N20050303;jvm=sun`
    - `-Declipse.perf.assertAgainst=build=N20040625`

Three variations: platform, build, jvm
Performance results are compared against reference data from previous releases.

- Problem: reference data not available for newly written tests

- Solution
  - Regenerate reference data on a weekly basis (or if new performance tests have been added)

- Requires
  - Branch for performance tests against previous releases
  - New tests must be ported back to reference branch
  - Only port if
    - performance test and its results are comparable across releases
    - measured functionality existed previously
Viewing Results Locally

- **Graphic tools/scripts available in Release Engineering project**
  - `org.eclipse.releng.basebuilder`
    - `org.eclipse.performance.test.ui.Main` generates fingerprints, data tables and line graphs

- **Performance plug-in provides simple “View” class for listing data as a table**
  - `(org.eclipse.test.internal.performance.db.View)`

- **You need to specify:**
  - Database location
  - Performance Scenario ID pattern (e.g. “%EditorTest%”)
  - Variation patterns (e.g. “platform=win32” and “build=I%”)
  - Tag for x-axis (e.g. “build”)

<table>
<thead>
<tr>
<th>Scenario: org.eclipse.jdt.text.tests.performance.RevertJavaEditorTest#testRevertJavaEditor()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builds:</td>
</tr>
<tr>
<td>CPU Time:</td>
</tr>
<tr>
<td>Elapsed Process:</td>
</tr>
<tr>
<td>Kernel time:</td>
</tr>
<tr>
<td>Page Faults:</td>
</tr>
<tr>
<td>System Time:</td>
</tr>
</tbody>
</table>
Interpreting Results

- Observe performance results regularly
  - Example: typing performance in Java Editor

- Performance results don’t pinpoint the problem
  - But they help to find problems early
  - This makes it easier to correlate a performance problem with other changes that are the likely cause

- Increased performance test coverage and performance tests on all application layers allow to further narrow down the origin of regressions
Advanced Usage

- Combine PerformanceMeter with full fledged profiler
  - Use profiler API to start/stop profiling
  - Reproduces the exact same scenario while collecting profiling data
  - Beware of distortions: the profiling method typically has an impact on the results

- Create your own PerformanceMeter
  - Example: count number of calls to specific paint method via JDI
Pitfalls

- Don’t make tests too short
  - On some platforms timer resolution is a few milliseconds (10ms on Win32)
  - Run them in a loop to bring normal run time in the range of 1s
    (beware of better JIT optimization)

- Be aware of startup costs (JIT, cache, etc.)
  - If interested in startup time
    - run test only once in JVM session (“Session Tests”)
    - use multiple JVM sessions to make average more stable
      (if collected data uses same tag, it is automatically aggregated)
  - If not interested in startup time
    - don’t collect data for first runs, use warm-up runs

- Keep things comparable
  - Compare with respective default preferences vs. compare with same preferences
Conclusion

- The Eclipse performance plug-in is not the “silver bullet” to performance problems
  - You still need a lot of creativity to find the real cause of a performance problem

- However, continuous performance testing makes it easier to pinpoint the cause because
  - you can spot a performance problem as soon as it occurs
  - it becomes easier to understand what code changes had occurred at the same time
Acknowledgments

- Performance infrastructure started by Text team
- IBM contributed part of implementation
- All other “guinea pigs” within OTI Labs

References

- Performance Tests How-To:
  [Performance Tests How-To](http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.test.performance/doc/Performance Tests HowTo.html)

- Performance tests for further illustration can be found, for example, in the `org.eclipse.jdt.text.tests` and other `*.tests` plug-ins