

# Problem Determination of your WebSphere Applications with RAD

Anita Rass Wan, RAD Product Manager



## Agenda

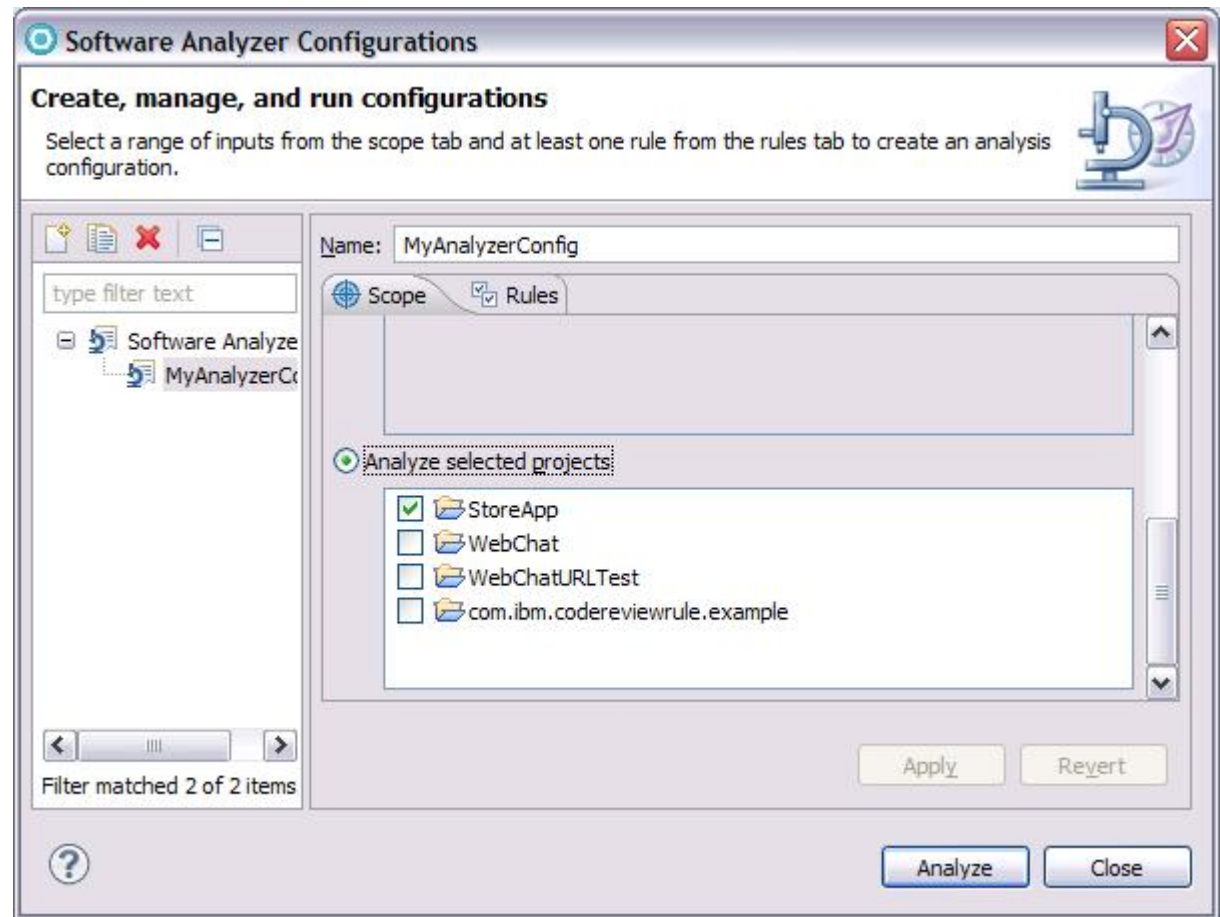
- Governance of Code
  - Creating your own rules - customizing rule templates
  - Advanced Rule creation
  - Development process integration
  - Best Practices
- Code coverage and Unit test optimization
  - Overview
  - Development process integration
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- Tuning the JVM
- Tuning WebSphere Applications
- Questions

## Static Analysis in Rational Application Developer 8.0

- Java Code Review to find problems of various types:
  - Design Principles
  - Globalization
  - J2EE & J2SE Best Practices
  - J2EE & J2SE Security
  - Naming
  - Performance
  - Private API
- Rich integrated results view provides click to source navigation
  - Explanations, examples, and quick fixes for problems
- Allow users to create, enable and disable validation rules
- Allow users to create their own rules based on rule templates
- Produce HTML/PDF reports with violations and metrics
- Complete Java Code Review (200+ rules)

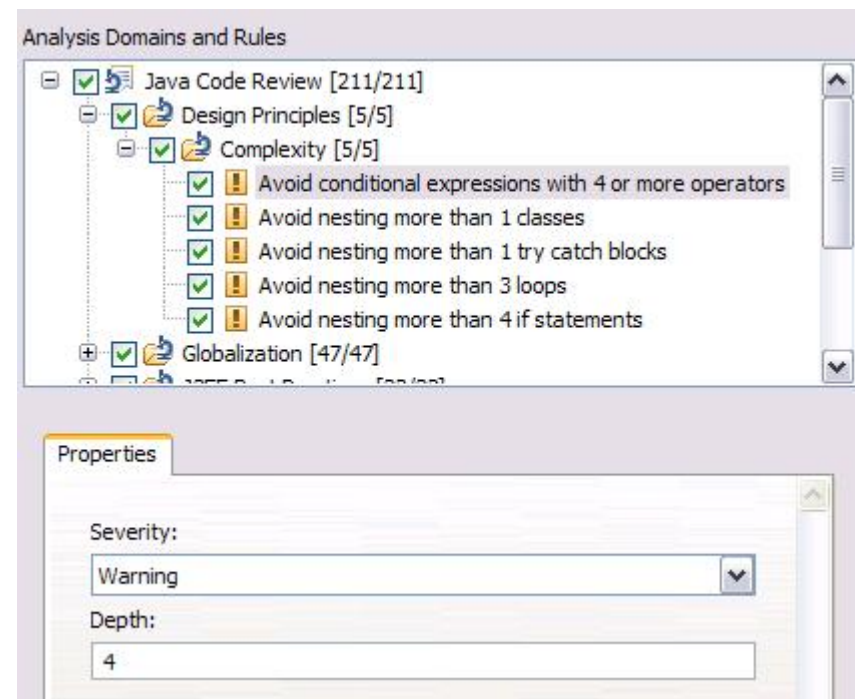
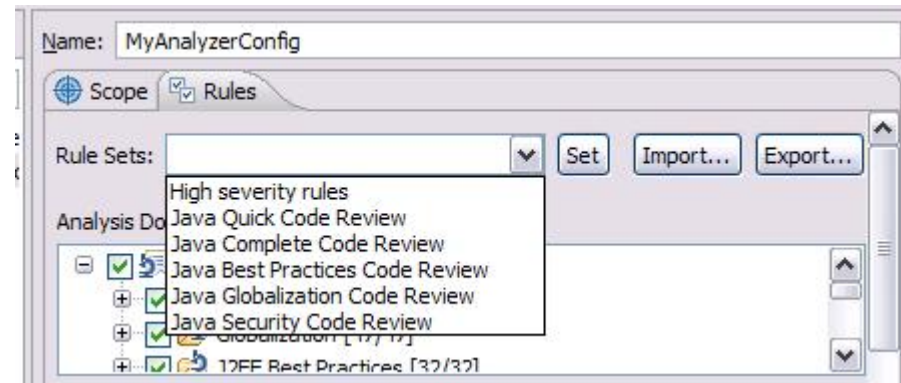
## Getting Started with Static Analysis

- Launched through Software Analyzer configuration
- Can create multiple configurations
  - Scope – workspace, working set or selected projects
  - Select and configure rules – focus on a particular aspect for analysis



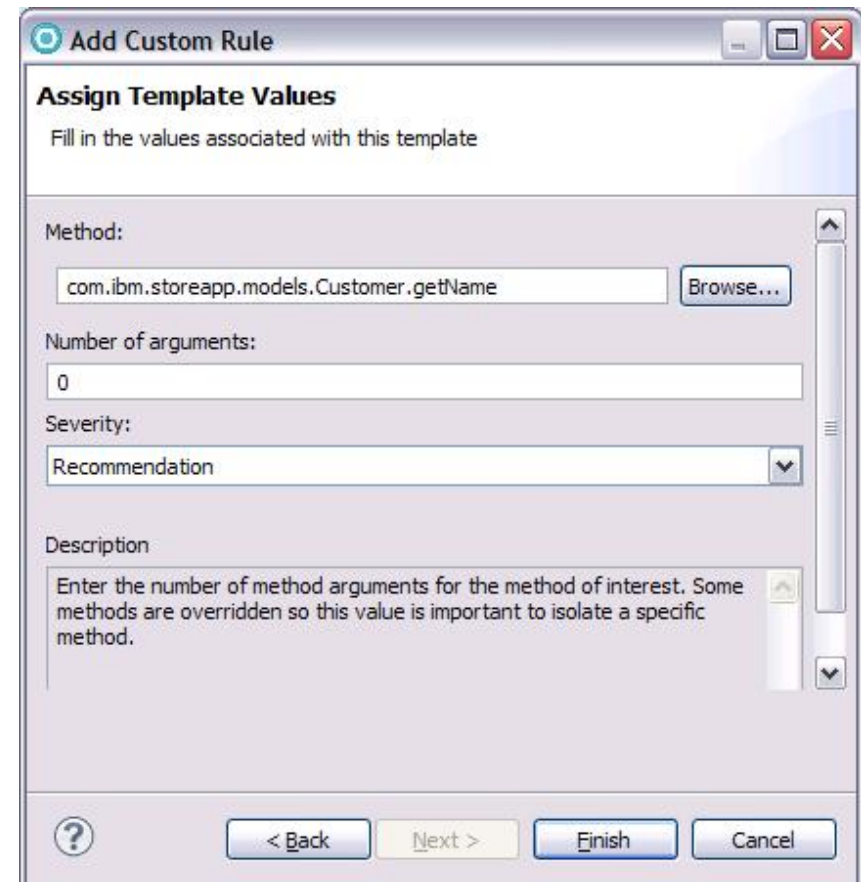
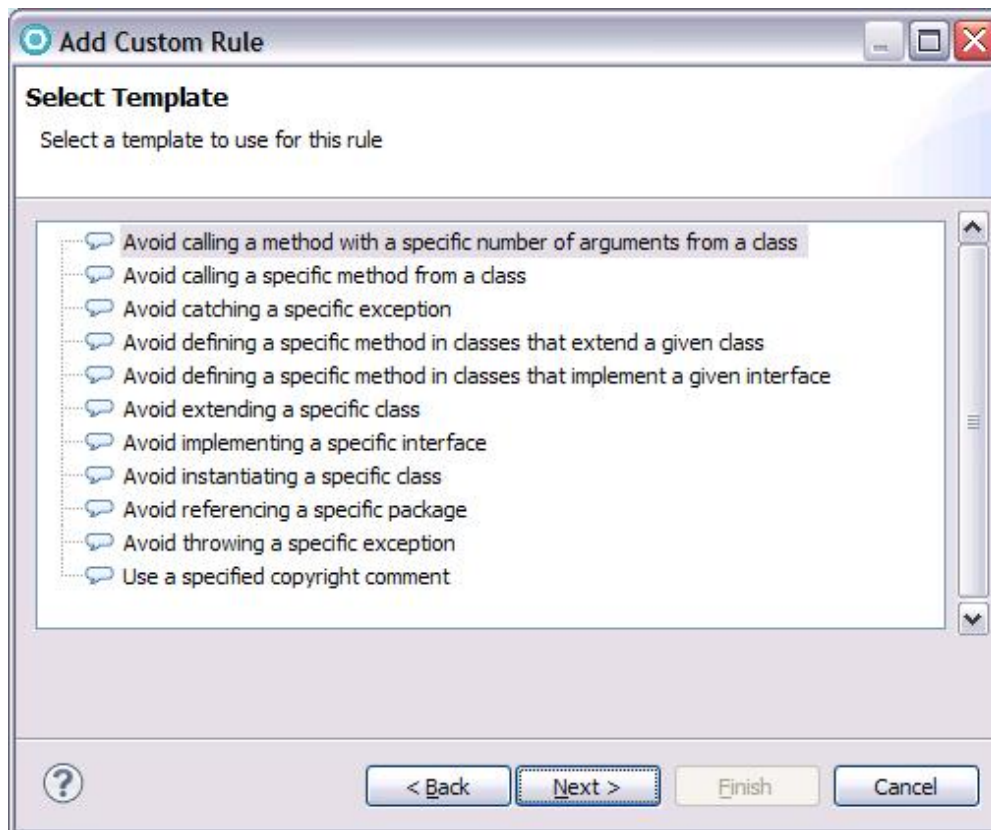
## Analysis Configuration – Rules

- Rule sets
  - Grouping of selected and configured rules that are to be run against the resources in the scope of the analysis
- Custom rule sets
  - Shared through export and import
- Rule configuration
  - Select which rules should be used in the current configuration
  - Configure severity of the violations
  - Change parameters of rules



## Adding Custom Rules

- Static analysis component comes with several templates that can be used to define custom rules through the preferences



## Advanced Rule Creation

- Why?
  - Sometimes the prepackaged or template based rules aren't specific enough
  - Domain specific constraints' or company wide governance enforcement rules
  
- How?
  - Software Analyzer component is extensible
  - Can create new categories/subcategories for your new rules
  - API framework makes it easier to implement
    - Rules, quickfixes, and even custom reports
  
- What?
  - Need to have some background in
    - Eclipse
    - Java
    - Eclipse JDT for Java rules

## Example – Rule to check for unused imports

- New category to hold our rule
- New rule definition as part of the new category
- Extend the abstract class `AbstractCodeReviewRule`
  - Implement `analyze()`
- Optionally implement an associated quickfix by extending `JavaCodeReviewQuickFix`
  - Implement `fixCodeReviewResult()`



## Analyzing Results

The screenshot shows an IDE window titled 'Store.java' with the following code:

```
private static void testCustomers() {  
    // creating 10 customers  
    int numCustomers = 10;  
  
    Store store = new Store(1);  
  
    // creating customers  
    for (int i = 0; i < numCustomers ; i++) {  
        store.addCustomer(new Customer("Customer " + i ));  
    }  
  
    // fetching customers
```

Below the code editor is a 'Java Code Review' panel. It shows a tree view of analysis results:

- Globalization [7 results in 81ms]
- J2SE Best Practices [21 results in 183ms]
- Performance [16 results in 54ms]
- Memory [6 results in 12ms]
  - Avoid java.lang.String + operator in loops [6 results in 0ms]
    - Store.java:45 Avoid java.lang.String + operator in loops
    - Store.java:57 Avoid java.lang.String + operator in loops
    - TestCustomer.java:19 Avoid java.lang.String + operator in loops

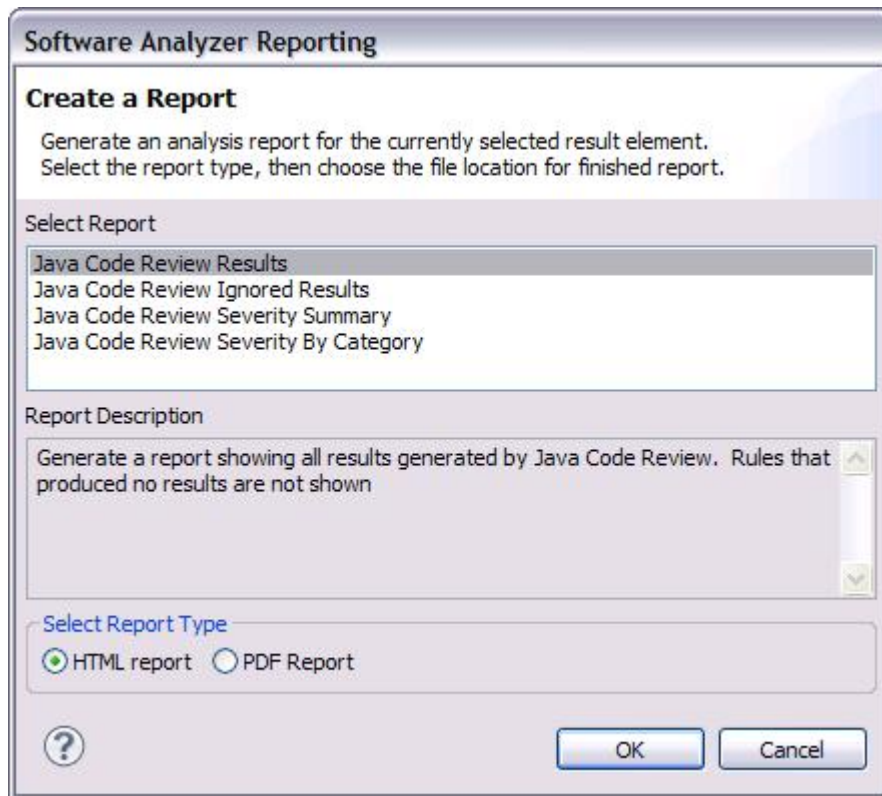
Export results in XML for other reporting tools

Generate reports HTML/PDF reports.

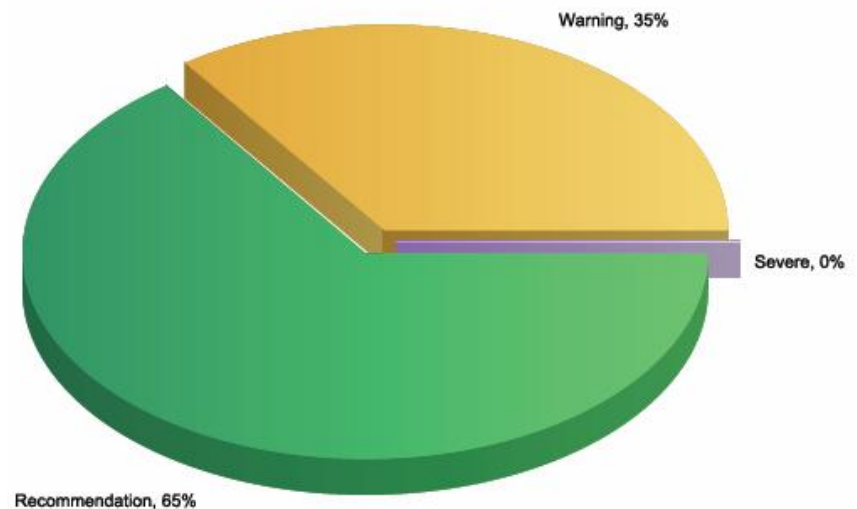
Double click and view the violation in the source view.

## Software Analyzer Reports

- Create HTML/PDF reports of different metrics
  - Results, Ignored Results, Severity Summary, Severity by Category



Java Code Review Severity Summary

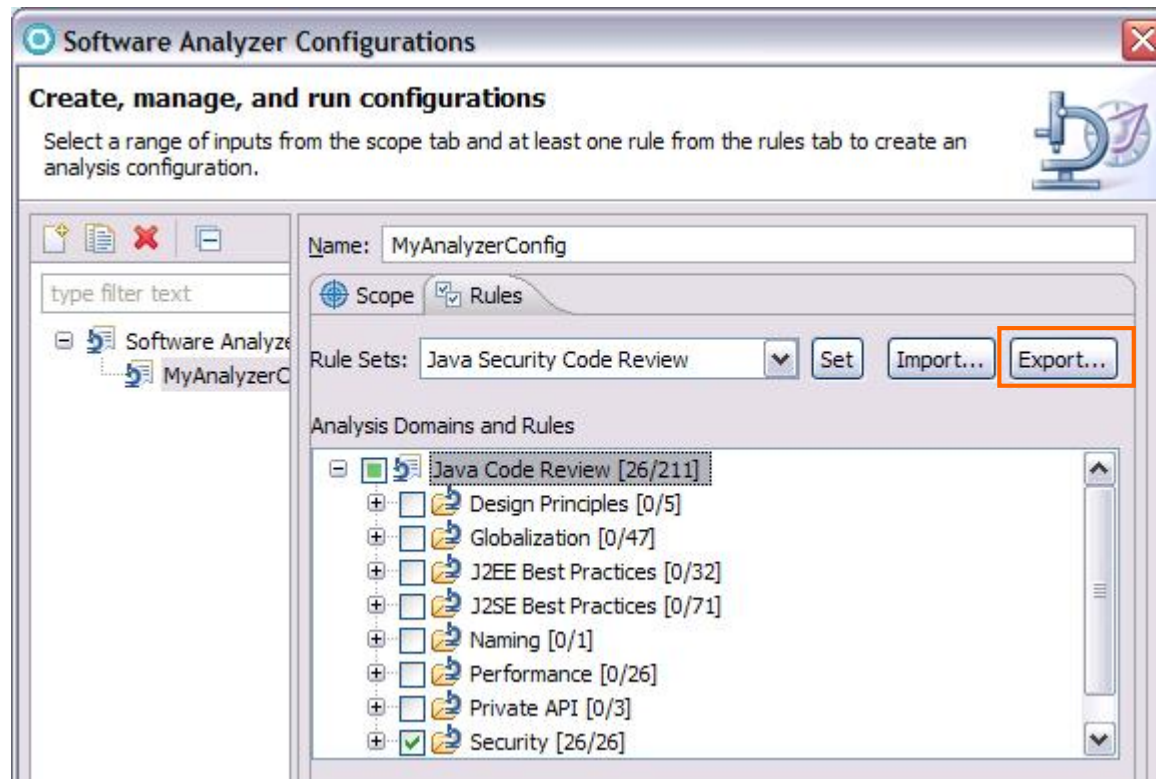


## Best Practices

- Run Analysis early and often
  - As part of developer best practice to run periodically
  - Cost to fix defects rises the further they get into the application's lifecycle
  - Logical separation of your application in terms of groups for analysis
  - Java code for business logic
  - Java code for connection services
  - Java code for presentation
- Smaller groups of analysis to avoid overwhelming number of results
  - Especially when first beginning to use the code review capability
  - Can increase the scope of analysis as code quality improves
- Create multiple rule sets to apply to different group types
  - Globalization rules apply more for java code involved in presentation
  - Different requirements for code complexity enforcement for business logic vs presentation tier

## Integrating Static Analysis Code Reviews – Builds – Step 1

Export your rule set (\*.dat) from the Analyzer Configuration



## Integrating Static Analysis Code Reviews – Builds – Step 2

Add a step to your build process to invoke the analysis.

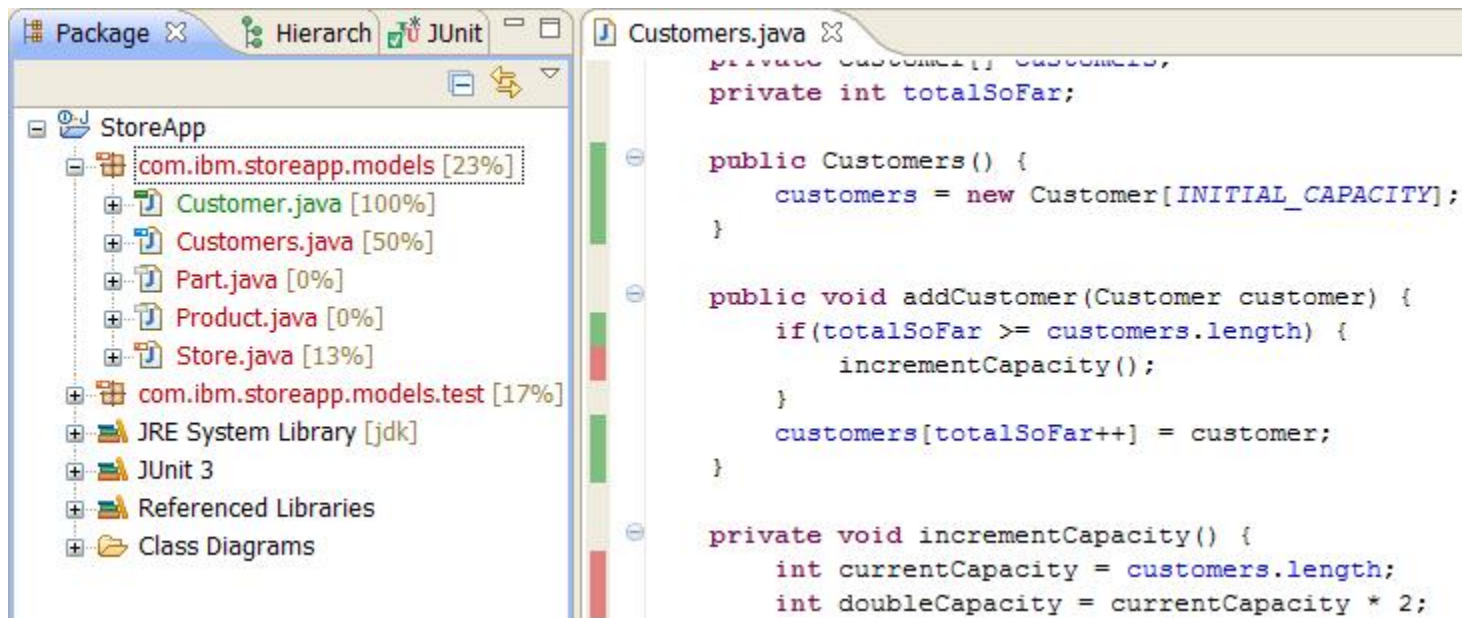
```
set
  WORKSPACE="C:\PROGRA~1\IBM\TeamConcertBuild\buildsystem\buildengine\eclipse\buil
  dDir"
set ECLIPSE="C:\PROGRA~1\IBM\SDP_1"
%ECLIPSE%\eclipse.exe
  -application com.ibm.xtools.analysis.commandline.AnalyzeApplication
  -rulefile "d:/rules.dat"
  -data %workspace%
  -directory %workspace%
  -exportdirectory %WORKSPACE%\RADcodeReviewXML
```

## Agenda

- Static Analysis of Code
  - Creating your own rules - customizing rule templates
  - Development process integration
  - Best Practices
- Code Coverage
  - Overview
  - Development process integration
- Advanced Code Review
  - Creating your own rules – write your own with the Software Analyzer API
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- JVM tuning
- Tuning WebSphere Applications
- Questions

## Code Coverage

- Provides detailed information on what code paths have been encountered during program execution
- Powerful tool to help determine xUnit test coverage, potential dead code
- Command line and Ant capability for build integration
  - JUnit, code coverage data collection and html report generation



The screenshot displays an IDE interface with two main panes. The left pane, titled 'Package Explorer', shows a project named 'StoreApp'. Underneath, there are two packages: 'com.ibm.storeapp.models' with a coverage of 23% and 'com.ibm.storeapp.models.test' with 17%. The 'com.ibm.storeapp.models' package contains several classes: 'Customer.java' (100%), 'Customers.java' (50%), 'Part.java' (0%), 'Product.java' (0%), and 'Store.java' (13%). The 'com.ibm.storeapp.models.test' package contains 'JRE System Library [jdk]', 'JUnit 3', 'Referenced Libraries', and 'Class Diagrams'. The right pane, titled 'Customers.java', shows the source code of the 'Customers' class. The code includes private fields for 'customers' and 'totalSoFar', a constructor, an 'addCustomer' method, and an 'incrementCapacity' method. A vertical bar on the left side of the code editor indicates the execution status of each line, with green representing executed code and red representing unexecuted code.

```
private Customer[] customers;
private int totalSoFar;

public Customers() {
    customers = new Customer[INITIAL_CAPACITY];
}

public void addCustomer(Customer customer) {
    if(totalSoFar >= customers.length) {
        incrementCapacity();
    }
    customers[totalSoFar++] = customer;
}

private void incrementCapacity() {
    int currentCapacity = customers.length;
    int doubleCapacity = currentCapacity * 2;
```

- Share code coverage information from automated testcase execution
- Improve test coverage and quality based on code coverage results

Code Coverage Report (Apr 26, 2010 11:14:54 AM)

Code Coverage Summary

Code coverage report for 'StoreApp', generated Apr 26, 2010 11:14:54 AM

Element	Coverage	Covered Lines	Total Lines
[StoreApp] com.ibm.storeapp.models.test	76%	11	69
[StoreApp] com.ibm.storeapp.models	24%	21	89
Store.java	12%	7	54
Product.java	0%	0	6
Pert.java	0%	0	8
Customers.java	52%	8	15
Customers	57%	8	15
addCustomer(): void	0%	0	6
Customer[]	100%	1	1
customers()	100%	3	3
addCustomer(Customer): void	88%	1	5
	100%	6	6

```

public void addCustomer(Customer customer) {
    if(totalSoFar >= customers.length) {
        incrementCapacity();
    }
    customers[totalSoFar++] = customer;
    for (int i = 0; i < 5000; i++) {

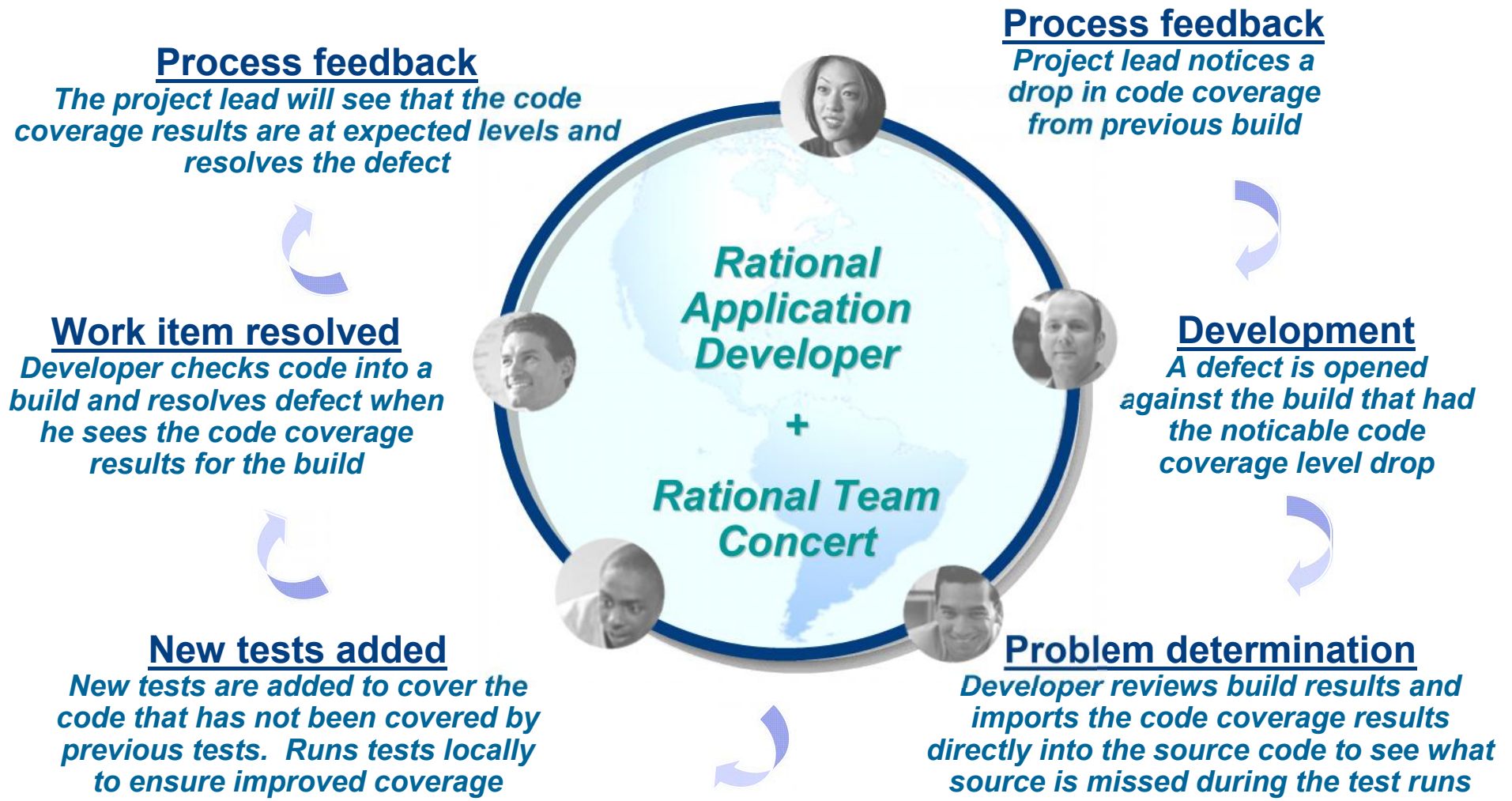
```

Lines 18 to 20: Covered



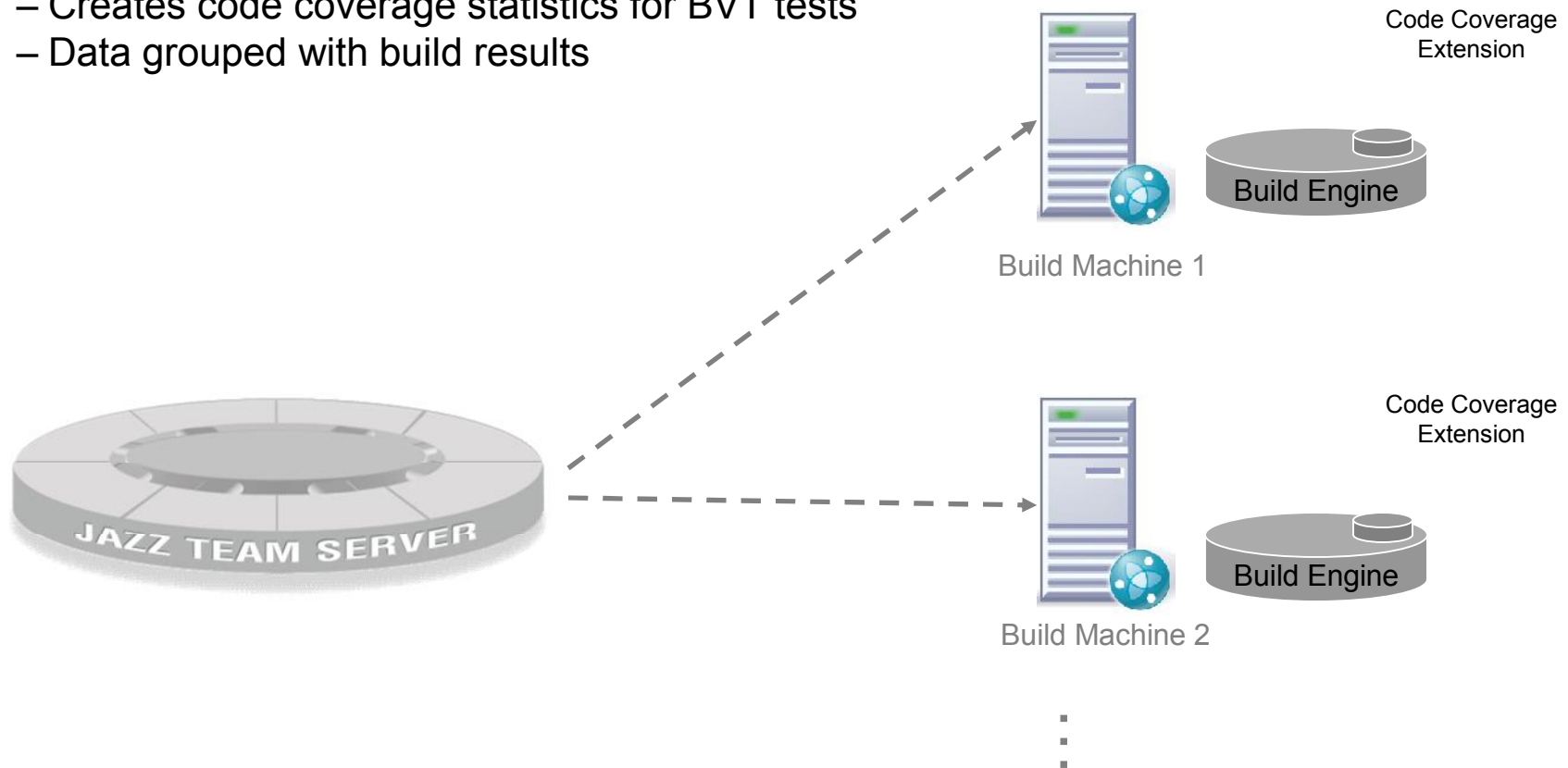


# Feedback metrics help identify shortfalls in test coverage



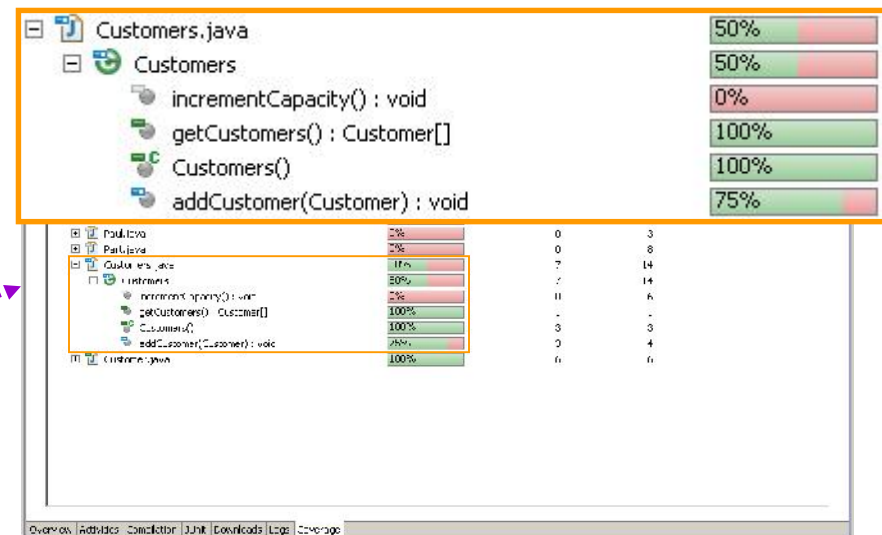
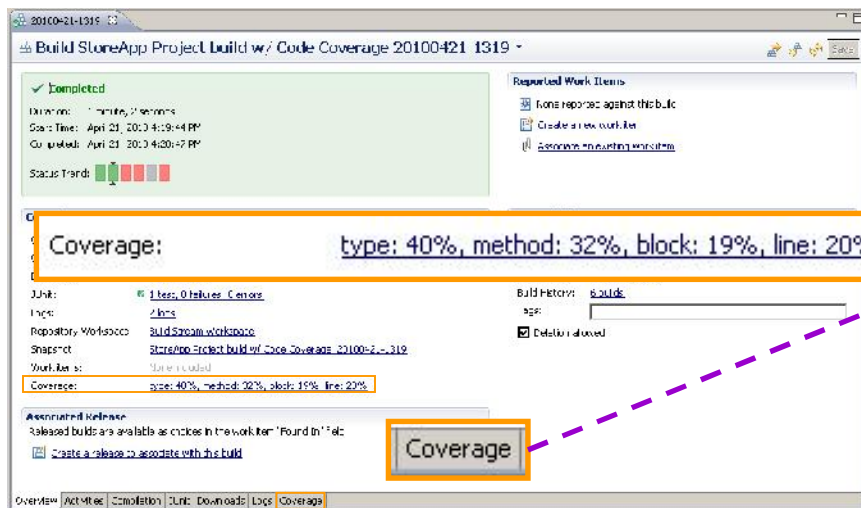
# Code Coverage Integration with RTC

- Build system integration
  - Installable RTC Build System extension
  - Creates code coverage statistics for BVT tests
  - Data grouped with build results



# Code Coverage Integration with RTC

- Client side
  - RTC Build details viewer extension – RAD and Web browser
    - Show summary of code coverage statistics
    - Additional Code Coverage tab to show detailed coverage statistics report
  - Integrated work item search and creation
  - Import to local workspace for rich viewing in navigator and source views



[Demo](#)

## Agenda

- Governance of Code
  - Creating your own rules - customizing rule templates
  - Advanced Rule creation
  - Development process integration
  - Best Practices
- Code coverage and Unit test optimization
  - Overview
  - Development process integration
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- Tuning the JVM
- Tuning WebSphere Applications
- Questions

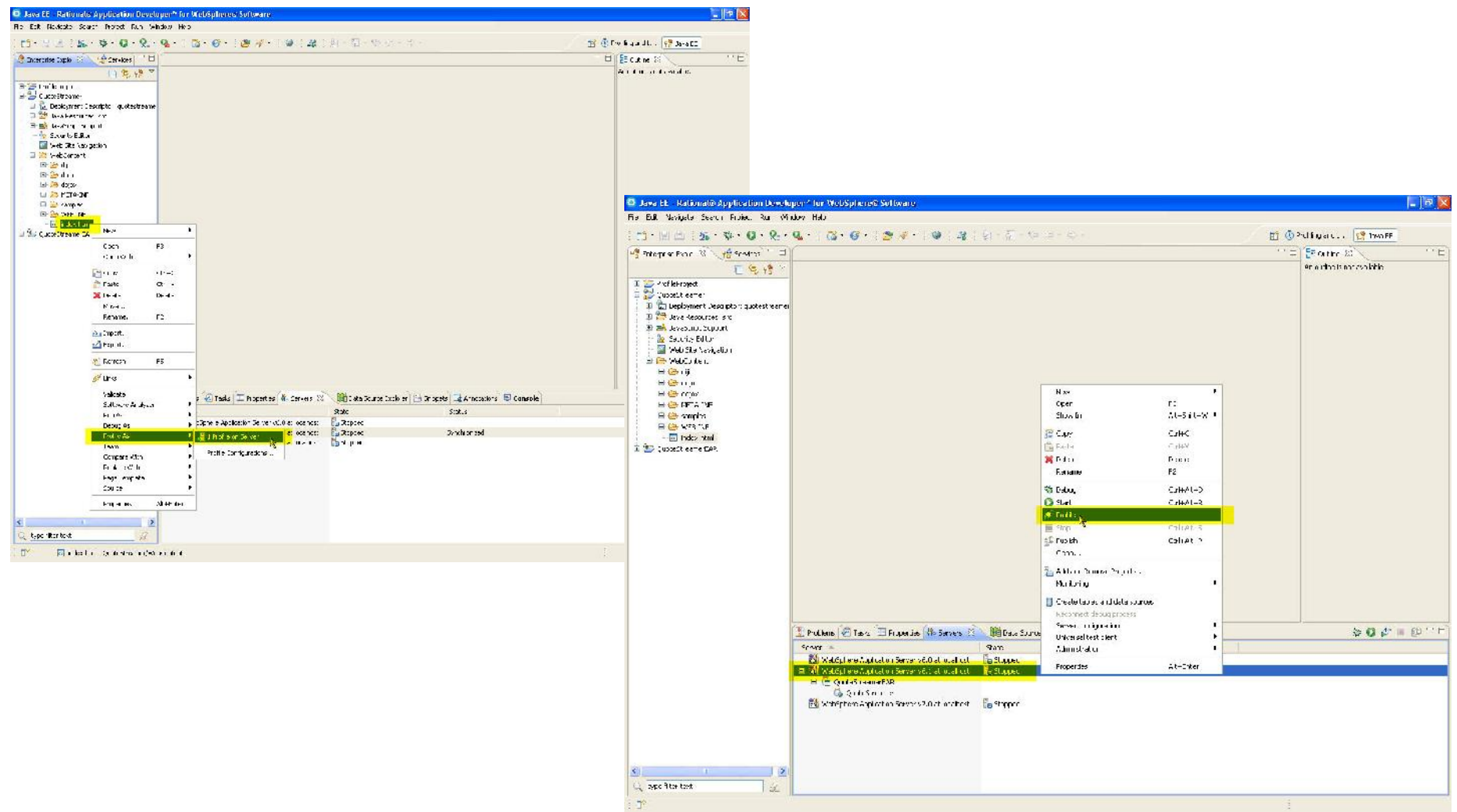
## What is Profile on Server?

- Profile on Server is the ability to look at various performance aspects of about your Java application
- Similar to Run and Debug on Server.
- Requires an Agent Controller (AC) running on the target machine to connect to the JVM
- Users start the profile action from a web resource or a server instance in Server Views.

## Agent Controller

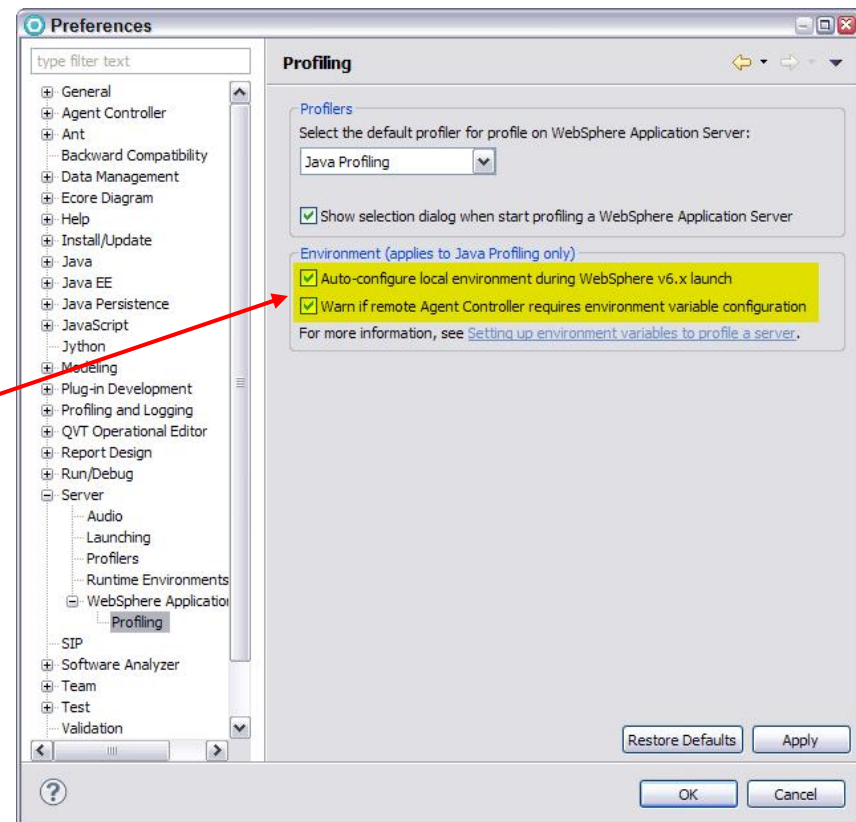
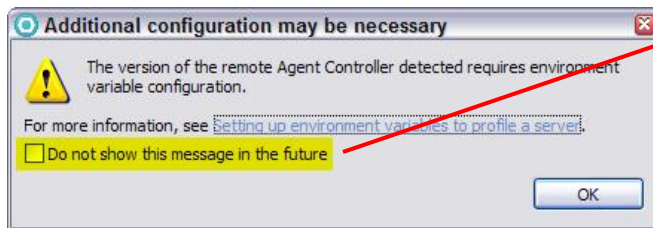
- A daemon process that enables client applications to launch host processes and interact with agents that coexist within host processes.
- An Agent Controller is required to be running on the target machine being profiled
  - ▶ Integrated Agent Controller (IAC) is embedded in RAD for local use case.
    - No additional installation,
    - Automatically enabled.
    - Start and stop on demand.
  - ▶ Standalone Rational Agent Controller ( RAC) is required for remote profiling.
    - Requires additional installation on target WAS machine.

# Launch profile on server



## New in RAD v8 – Easy profiling setup

- If using an older RAC on a remote machine, RAD will auto detect the older version and prompt to add environment variables or upgrade your RAC





## Quick start Wizard




### Profiling Quick Start Wizard

#### Method of Profiling

Here we will select the type of profiling data to collect, based on the type of problem that needs solving.



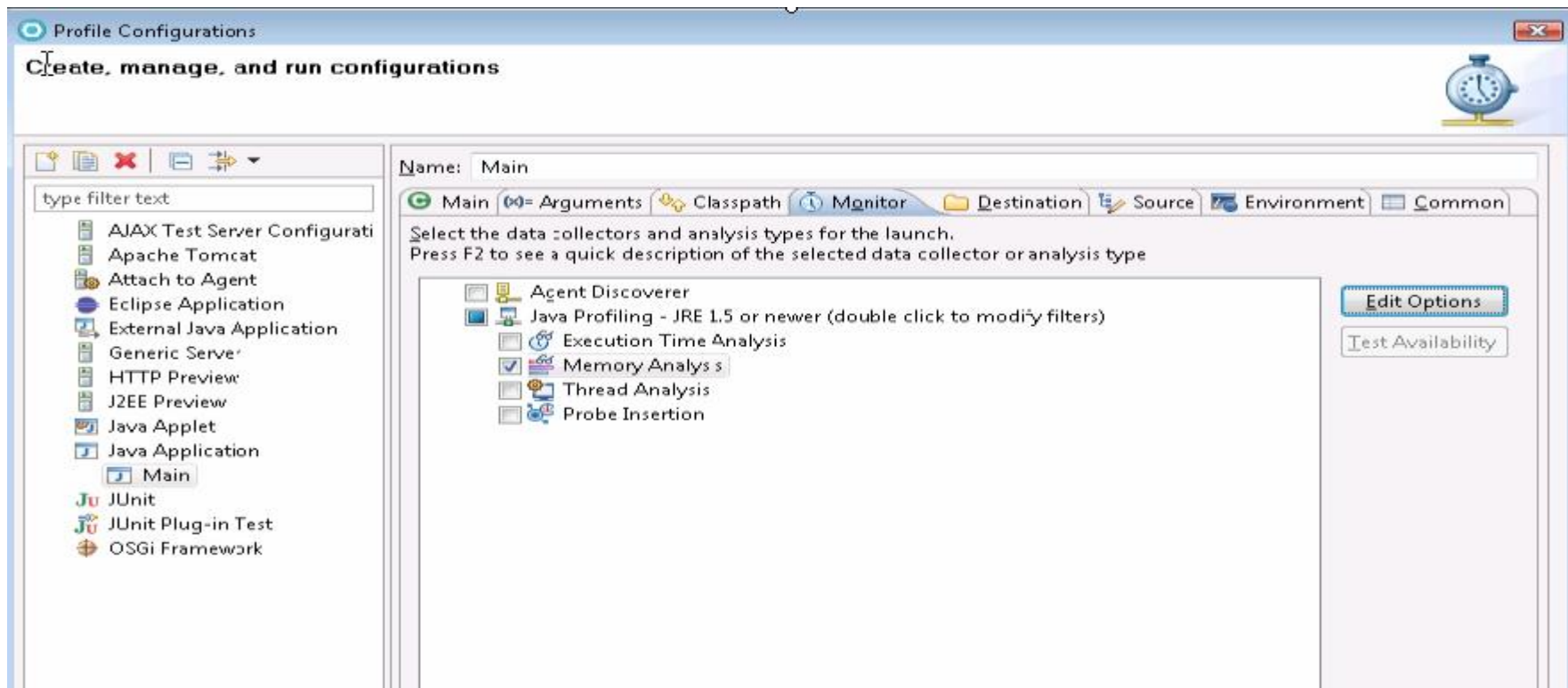
Profiling allows you to identify and eliminate performance problems. Select the performance issue below:

-  My application is slow  
There's a performance bottleneck that is causing transaction throughput to be limited, or is hurting application response time.
-  My application takes up too much memory  
There's a memory leak causing out of memory errors or inflating application size. The amount of memory used in application sessions is too high causing the number of concurrent sessions to be limited.
-  My application experiences long pauses independent of CPU activity  
There are long pauses or hangs in the application that do not seem to be related to processor usage, but rather to thread contention, such as blocked threads or race conditions.

## Select Memory Analysis Option

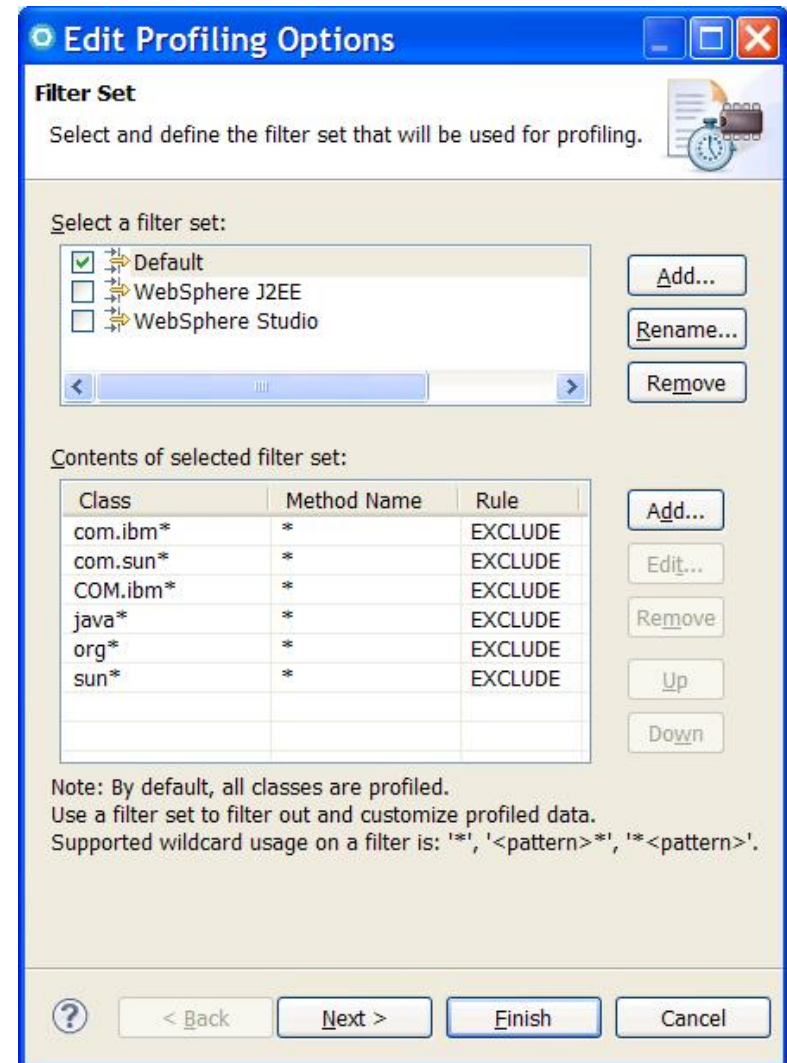
Instrumentation available for profiling Java applications:

- Execution Time Analysis
- Memory Analysis
- Thread Analysis
- User written probes



## Profiling Options

- Data collection can be expensive
  - Application under test slows down
  - Large amount of data for both man and machine
- Filter Sets
  - Collect data of interest
  - Focus on your code
  - Can be exported for sharing



## Profiling Metrics

- Determine application behavior
  - Execution time, number of calls, call tree

Thread name	<Percent Per Thread	Cumulative...	Min Time	Avg Ti...	Max Time	Calls
main[9696]	100.00%	2.381987				
↳ TestSuite(java.lang.Class)	26.88%	0.640376	0.640376	0.640376	0.640376	1
↳ addTestMethod(java.lang.reflect.Met	25.93%	0.617642	0.000046	0.051470	0.616993	12
↳ createTest(java.lang.Class, java.	25.89%	0.616682	0.616682	0.616682	0.616682	1
↳ -clinit-()	9.25%	0.220281	0.220281	0.220281	0.220281	1
↳ staticInitializer/java.lang.Stri	0.77%	0.018303	0.018303	0.018303	0.018303	1

- Memory allocation

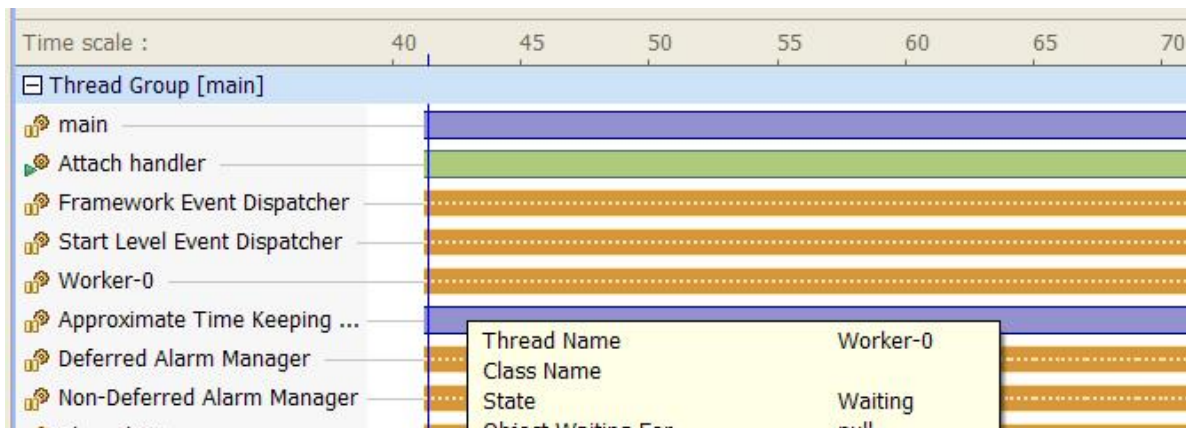
Class Name	Package	Live Inst...	Active Size...	Total Instances	<Total Siz...	Avg. Age
byte[]	(default package)	230	171808	1457	963528	0.08
char[]	(default package)	91	68544	301	114848	0.04
int[]	(default package)	394	21112	731	50664	0.48
long[]	(default package)	3	2232	3	2232	0.67
short[]	(default package)	2	1568	2	1568	0.5
boolean[]	(default package)	9	1456	12	1552	0.75
int[][]	(default package)	0	0	3	120	0

## Thread metrics

- Thread Statistics

Thread Name	Class Name	State	<Running Time	Waiting Time	Blocked Time	Block Count	Deadlocked Time	Deadlock Count
P=243359:O=0:WST		Running	00:44:036					
P=243359:O=0:WST		Running	00:44:036					
ORB.thread.pool : 0		Waiting	00:00:075	00:43:961				
ORB.thread.pool : 1		Waiting	00:00:073	00:43:962				
Deferred Alarm Mana		Waiting	00:00:007	00:44:029				
Deferrable Alarm : 0		Waiting	00:00:005	00:44:030				
Non-Deferred Alarm I		Waiting	00:00:003	00:44:033				
Deferrable Alarm : 1		Waiting	00:00:003	00:44:032				
server.startup : 0		Stopped	00:00:002	00:13:033				

- Threads Visualizer – view timeline of threads with state



## Custom profiling

- Custom Profiling
  - Implement your own Java probes to collect runtime information from your application
  
- Java code fragments
  - Class variables
  - Runtime fields
    - thisObject
    - className
    - ...
  - Target filters

The screenshot shows a configuration window for a custom probe. It includes the following fields and options:

- File Name:** myCustomProbe.probe
- Source Folder:** /CustomProbe/src (with a **Browse** button)
- XML Encoding:** UTF-8 (dropdown menu)
- Add content to the probe file:**
  - Method Probe
  - Callsite Probe
  - No Content (Blank Probe File)
- Fragment types:**
  - catch
  - entry
  - executableUnit
  - exit
  - staticInitializer

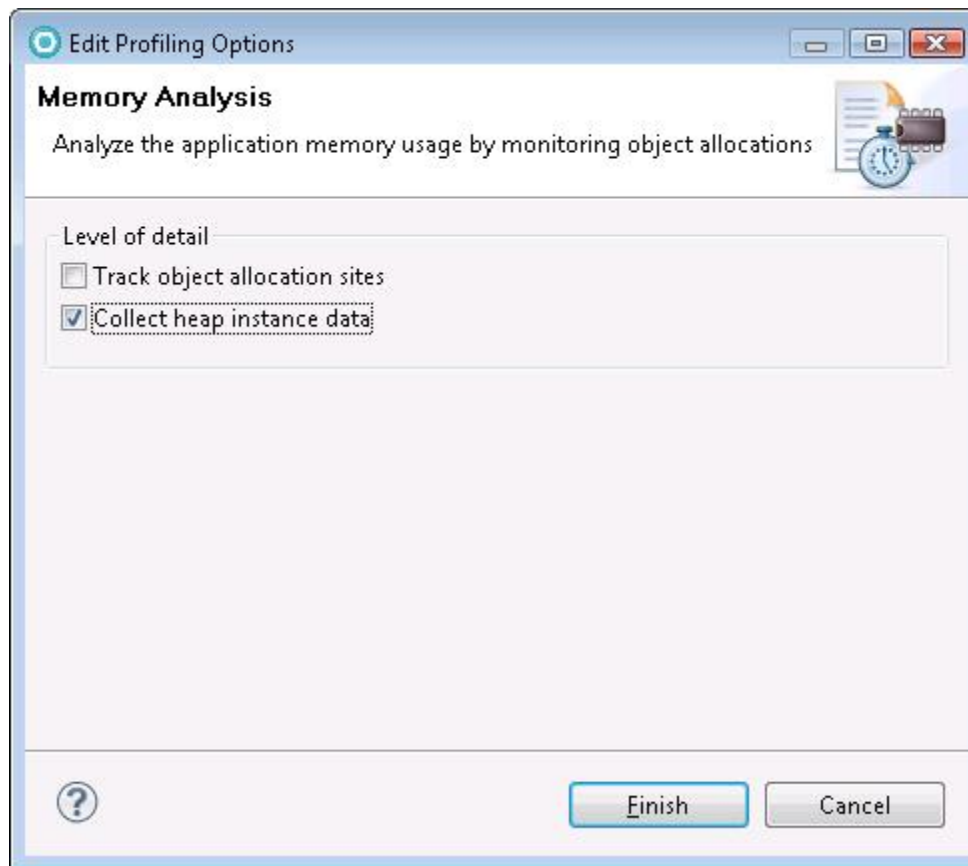
Additional text on the right side of the dialog:

- For **Method Probe**: This type of probe is inserted anywhere within the body of a method. For method probes, the class or jar files containing the target methods are instrumented by the byte-code instrumentation (BCI) engine.
- For **Fragment types**: entry fragments execute upon method entry. entry fragments will not execute for methods that were inserted into the class by Probekit.

## Memory Analysis

- Memory Analysis is a profiling analysis type for Java 1.5+ which displays collected data about Objects during application execution
- Instance data collection is a new option that can be enabled during Profiling. It provides the ability to inspect the composition of Objects, including obtaining the live Object values, variable names and instance size during the profile session.
- The data that has been collected can be exported to an XML file.

# Enable Instance Data Collection





- Can also import/export to XML file for comparison

Object Allocations ✕

Memory Analysis - live.instance.test.Main at tptp-jayhawk [ PID: 3544 ]

### Memory Statistics

Filter: No filter. [Click here to set filter](#)

>Class Name	Package	Live Instan...	Active Size ...	Total Insta...	Total Size (...)	Avg. Age
Keystore	live.instance.test	2	32	2	32	0
Keystore (id=418)			16		16	
Keystore\$Entry (id=422)			24		24	
String key = Key 1;						
String value = Value 1;						
Keystore (id=430)			16		16	
Keystore\$Entry	live.instance.test	151	3624	151	3624	0
[ 0 .. 99]						
Keystore\$Entry (id=422)			24		24	
String key = Key 1;						
String value = Value 1;						
Keystore\$Entry (id=487)					24	
Keystore\$Entry (id=488)			24		24	

Instance of live.instance.test.Keystore\$Entry (id=422), size=24

Press F2 to focus

## Agenda

- Governance of Code
  - Creating your own rules - customizing rule templates
  - Advanced Rule creation
  - Development process integration
  - Best Practices
- Code coverage and Unit test optimization
  - Overview
  - Development process integration
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- Tuning the JVM
- Tuning WebSphere Applications
- Questions

## IBM Support Assistant

- Single access point for finding and downloading support tools for IBM products
  
- Also includes many tools for problem determination, including:
  - JVM Health Center
  - Thread and Monitor Dump Analyzer
  
- Access from RAD launch-pad

## Diagnostic Data

- Data from the JVM -
  - Javacore – thread information
  - Heapdumps – heap details
  - Verbose GC logging – garbage collection data
- Data from agents connected to the JVM
  - Health Center
  - Rational Agent Controller
  - More detailed information, higher accuracy: thread, heap details, execution flow
  - More overhead than Javacore or Heapdumps
- Data from the Application Server: PMI

## General Strategy

- Try to isolate issues using data that's easily available or obtainable:
  - Javacores, heapdump files
  - Use wsadmin to interactively send events to JVM MBean
  
- If problem not resolved with this level of information
  - Narrow area of investigation and use filtering combined with more detailed data collection agents
  
- There is often more than one tool that could be used
  
- Measure, measure, measure!

## Crashes

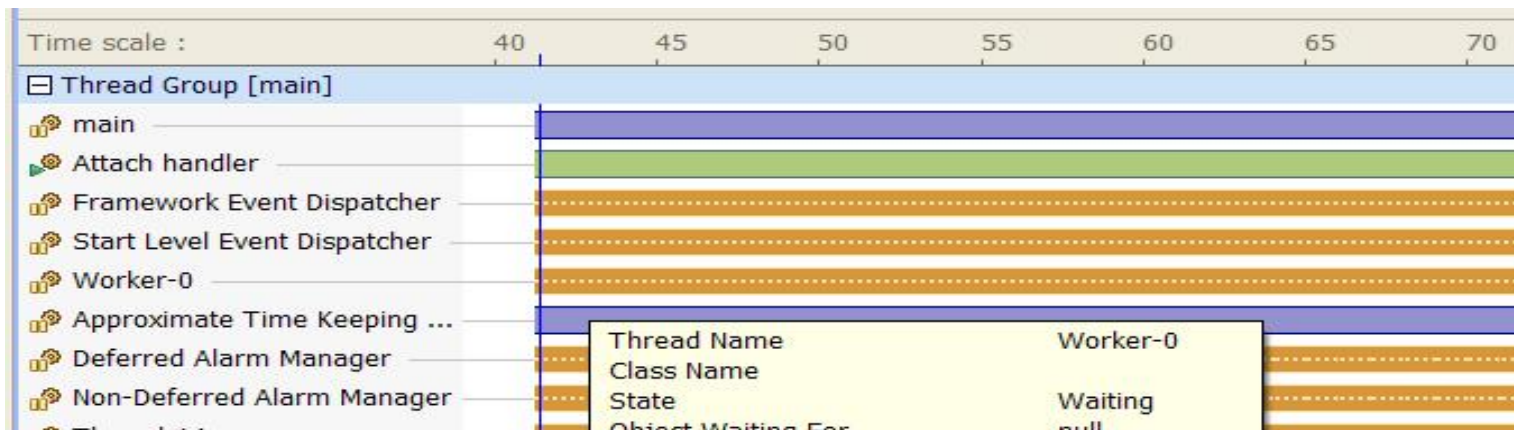
Problem: The untimely death of an application

- View WebSphere server logs
  
- Use Thread and Monitor Dump Analyzer
  - Javacore file from the application
  - Identifies where the threads last executed

## Hangs & Deadlocks

Problem: App is completely unresponsive

- Use Rational Application Developer
  - Complete hangs - run under debugger and pause after unresponsive point
  - Hangs/deadlocks
    - Thread Analysis for more detailed data
    - Thread statistics and Threads Visualization contention



- Use Thread and Dump Analyzer

## Memory Leaks / Excessive Memory Use

### Problem:

- Slowdown due to increased garbage collection
- Reduced number of concurrent sessions
- Use Rational Application Developer
  - Use Memory Analysis
  - Identify offending object types
  - Restart analysis, tracking object allocation sites if understanding where they're being allocated is needed
- Consider reducing session timeout to reduce longevity of session objects

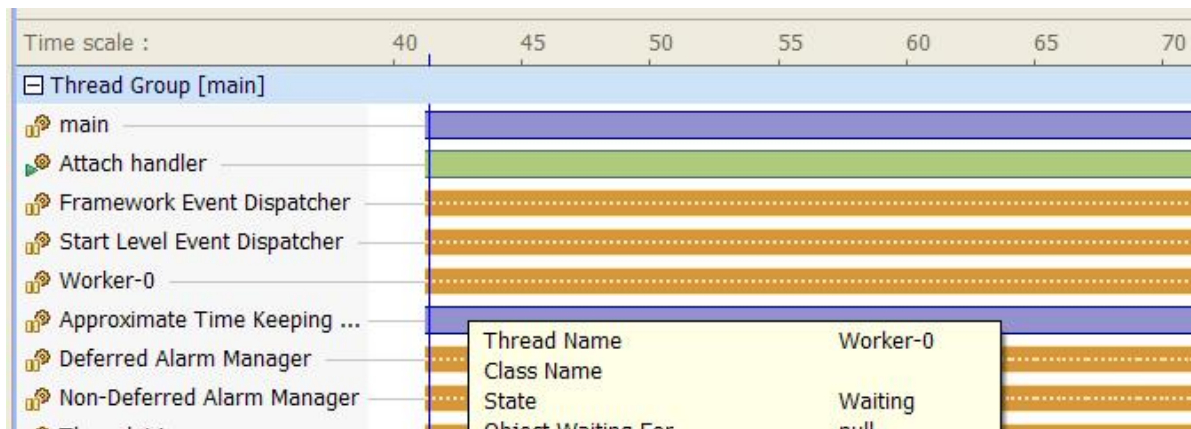
Class Name	Package	Live Inst...	Active Size...	Total Instances	<Total Siz...	Avg. Age
byte[]	(default package)	230	171808	1457	963528	0.08
char[]	(default package)	91	68544	301	114848	0.04
int[]	(default package)	394	21112	731	50664	0.48
long[]	(default package)	3	2232	3	2232	0.67
short[]	(default package)	2	1568	2	1568	0.5
boolean[]	(default package)	9	1456	12	1552	0.75
int[][]	(default package)	0	0	3	120	0



## Bottlenecks – Overly Aggressive Locking

Problem: slow responsiveness and throughput of web apps

- Potential cause: multithreaded code blocks protect more than they need
- Use Rational Application Developer
  - Thread Analysis for more detailed data
  - Thread statistics and Threads Visualization to view overlapping contention areas over time
  - Investigate long lock areas



[Demo](#)

## Bottlenecks – Inefficient Code Paths

Problem: Slow responsiveness and throughput of web apps

- Potential cause: redundant calls or long running code
- Use Rational Application Developer
  - Execution time analysis for detailed metrics
  - Look for long cumulative times and base times

Thread name	<Percent Per Thread	Cumulative...	Min Time	Avg Ti...	Max Time	Calls
main[9696]	100.00%	2.381987				
TestSuite(java.lang.Class)	26.88%	0.640376	0.640376	0.640376	0.640376	1
addTestMethod(java.lang.reflect.Met	25.93%	0.617642	0.000046	0.051470	0.616993	12
createTest(java.lang.Class, java.	25.89%	0.616682	0.616682	0.616682	0.616682	1
-clinit-()	9.25%	0.220281	0.220281	0.220281	0.220281	1
staticInitializer/java lang Stri	0.77%	0.018303	0.018303	0.018303	0.018303	1

[Demo](#)

## Agenda

- Governance of Code
  - Creating your own rules - customizing rule templates
  - Advanced Rule creation
  - Development process integration
  - Best Practices
- Code coverage and Unit test optimization
  - Overview
  - Development process integration
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- Tuning the JVM
- Tuning WebSphere Applications
- Questions

## JVM Tuning – Garbage Collection Strategy and Heap size

- Potential Issues:
  - Pause times – server unable to process requests due to GC pause times
  - Factors affected GC frequency and size/number of session objects supported
    - Garbage collection policies
    - Heap size parameters
  
- Use Health Center – live recommendations for GC strategy and heap size
  - Execute a reasonable portion of your application first
  
- Use Garbage Collection and Memory Visualizer
  - Enable verbose GC output
  - Provides more detailed tuning recommendations

[Demo](#)

## Agenda

- Governance of Code
  - Creating your own rules - customizing rule templates
  - Advanced Rule creation
  - Development process integration
  - Best Practices
- Code coverage and Unit test optimization
  - Overview
  - Development process integration
- Profiling
- Problem scenarios: What should I use?
  - Crashes, hangs
  - Memory leaks
  - Execution bottlenecks
- Tuning the JVM
- Tuning WebSphere Applications
- Questions

## WebSphere Application Server Tuning

- Server configuration can affect the performance of your application
  
- Analysis Tools
  - Enable logging of key PMI data via admin console
    - Open tivoli view – showing various tabs that can be monitored
    - Start logging activity for scenarios you want to optimize for
    - Use the performance advisor to produce tuning recommendations
    - Retry the scenario with configuration changes

[Demo](#)

## Conclusions

- Finding problems early in development saves money
- Code Coverage from Rational Application Developer running on a server
  - Verify what get's touched incrementally
  - Combine with unit tests or BVT to automate and track
- Many tools available to aid in problem determination:
  - Understanding what the various diagnostic data contains helps guide problem determination strategy for particular problems
    - Different tools operate on different input data
  - Detailed and accurate data collection incur overhead
    - Identify focus areas to filter data collection
  - WebSphere Application Server contains embedded performance monitoring
    - Useful Java EE centric performance data
    - Advisor provides server tuning suggests for you applications

## Copyright and Trademarks

© IBM Corporation 2011. All Rights Reserved.

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).



## Export Collected Object Data

- The data, such as in the example below, can be exported and used for comparison

```
<?xml version="1.0" encoding="UTF-8" ?>
<heap-instance-data version="1">
  ...
  <complex-obj tid="418" parent-tid="429" class="Keystore" package="live.instance.test" size="16">
    <object tid="423" parent-tid="418" class="Entry:entries" package="" size="24"/>
  </complex-obj>
  <complex-obj tid="422" parent-tid="423" class="Entry:next" package="" size="24">
    <primitive type="String" name="value">Value 1</primitive>
    <primitive type="String" name="key">Key 1</primitive>
  </complex-obj>
  <complex-obj tid="423" parent-tid="418" class="Entry:entries" package="" size="24">
    <object tid="422" parent-tid="423" class="Entry:next" package="" size="24"/>
    <primitive type="String" name="value">Value 2</primitive>
    <primitive type="String" name="key">Key 2</primitive>
  </complex-obj>
  ...
</heap-instance-data>
```