Detecting and Solving J2EE Performance Problems

Eyal Oz

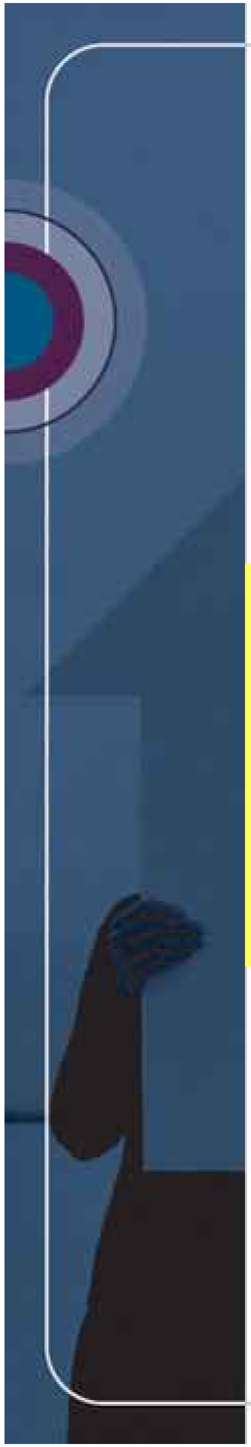
Director,

Load Testing & Tuning R&D

Mercury Interactive

Presentation Goal

Learn how to detect J2EE performance problems in your application and how to narrow down the root cause and solve it



The Motivation

“98% of all web sites have a

Bad User Experience!

only scaling to 25% of their design capability”

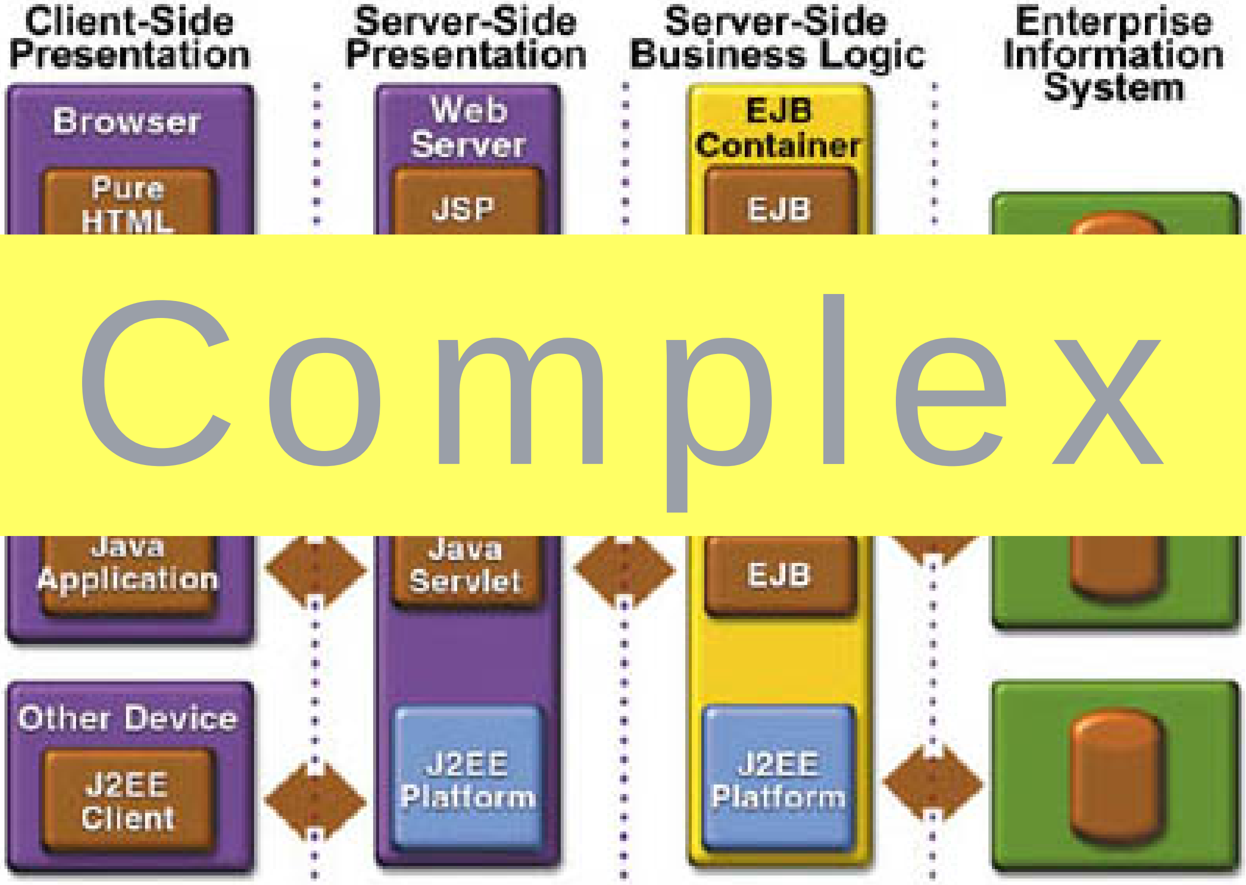
– ActiveTune service, Mercury Interactive

Presentation Agenda

- **Why** do J2EE applications have performance problems?
- **How** do I know if I have performance problems?
- **Where** in my application are the performance problems?
- **How** do I drill-down to the root cause?

- Summary
- Q&A

Why do J2EE Applications have Performance Problems ?



Complex !

The J2EE Application Challenge

- Based on multiple tiers, including Web-server, application-server and database – making the overall architecture complex
- Depend greatly on the vendor, mainly because of application-server container implementations
- Depend on the configuration of the different tiers
- Depend on the JVM/s running on the different tiers
- J2EE applications contain a lot of “distributed” logic, which needs to be carefully designed and implemented

The J2EE Application Challenge - A Few Examples

- Common application server problems
  - Poor cache management
  - Non-optimized database queries
  - Poor concurrent handling of client request
  - Frequent writes to directory server
  - Synchronization in Servlets which avoids multiple execution threads, becoming effectively single-threaded
  - Poorly designed algorithms

How do I know I have Performance Problems ?

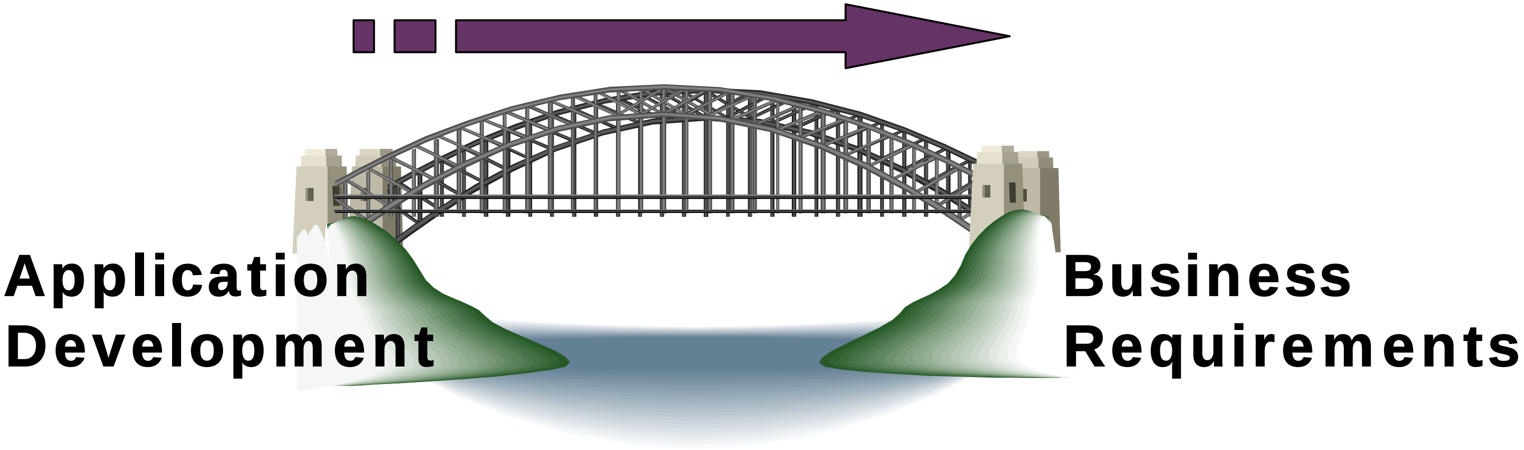
- A developer might say:

  “I tested my bean and it functions well…”

How Do We Bridge This GAP?

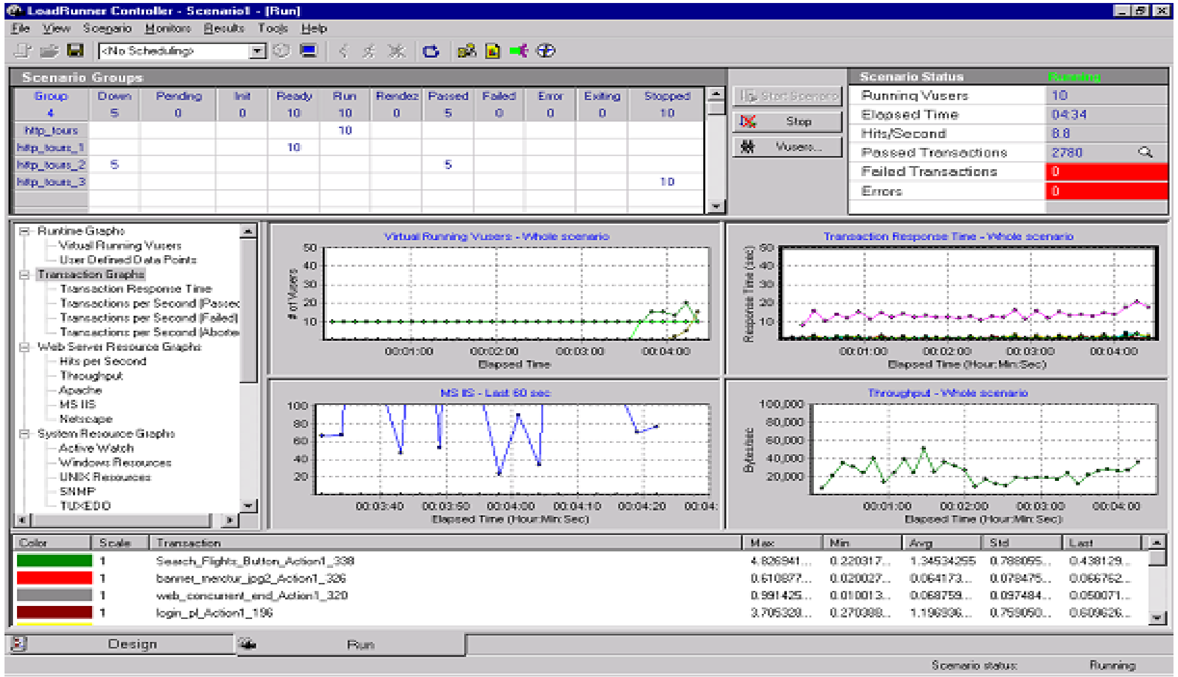
- But:
  - Did he test end-user experience?
  - Did he test the application with multiple users?
  - Does he know what are the critical paths in the application?
  - Did he invest any time in optimizing these paths?

Bridge the Gap With Testing



- Test your J2EE Application under real concurrent end-user conditions
- Monitor your entire J2EE stack
- Correlate end-user experience with your system status
- Drill down from the end-user experience to the different layers

Load Test Your J2EE Application



Scenario Groups

Group	Down	Pending	Init	Ready	Run	Rendez	Passed	Failed	Error	Exiting	Stopped
4	5	0	0	10	10	0	5	0	0	0	10
http_tours					10						
http_tours_1				10							
http_tours_2	5						5				
http_tours_3											10

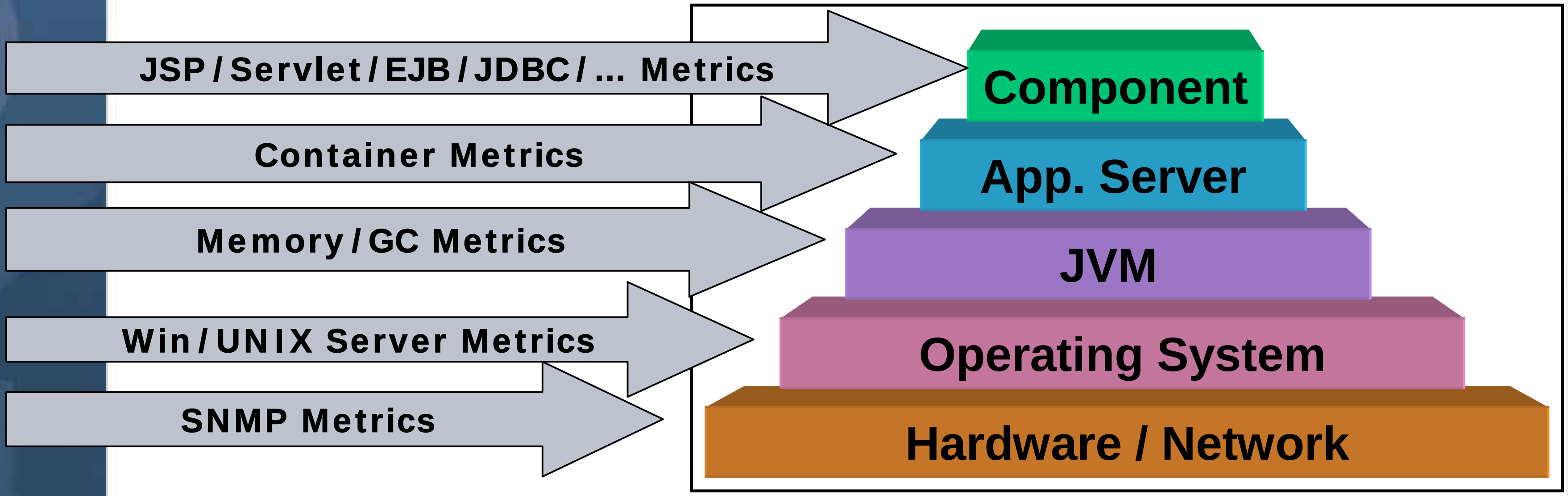
Scenario Status	Running
Running Vusers	10
Elapsed Time	04:34
Hits/Second	8.8
Passed Transactions	2780
Failed Transactions	0
Errors	0

Color	Scale	Transaction	Max	Min	Avg	Std	Last
🟩	1	Search_Flights_Button_Action1_338	4.826941...	0.220317...	1.34534255	0.788055...	0.438129...
🟥	1	banner_merctur_jpg2_Action1_326	0.610877...	0.020027...	0.064173...	0.079475...	0.066752...
▨	1	web_concurrent_end_Action1_320	0.991425...	0.010013...	0.068759...	0.097484...	0.050071...
🟥	1	login_pl_Action1_196	3.705328...	0.270388...	1.198936...	0.759050...	0.609626...

Scenario status: Running

- Use well-known load testing tools (like Mercury's LoadRunner)
- Emulate real life conditions
  - Emulate the WAN (if necessary)
  - Emulate peak loads

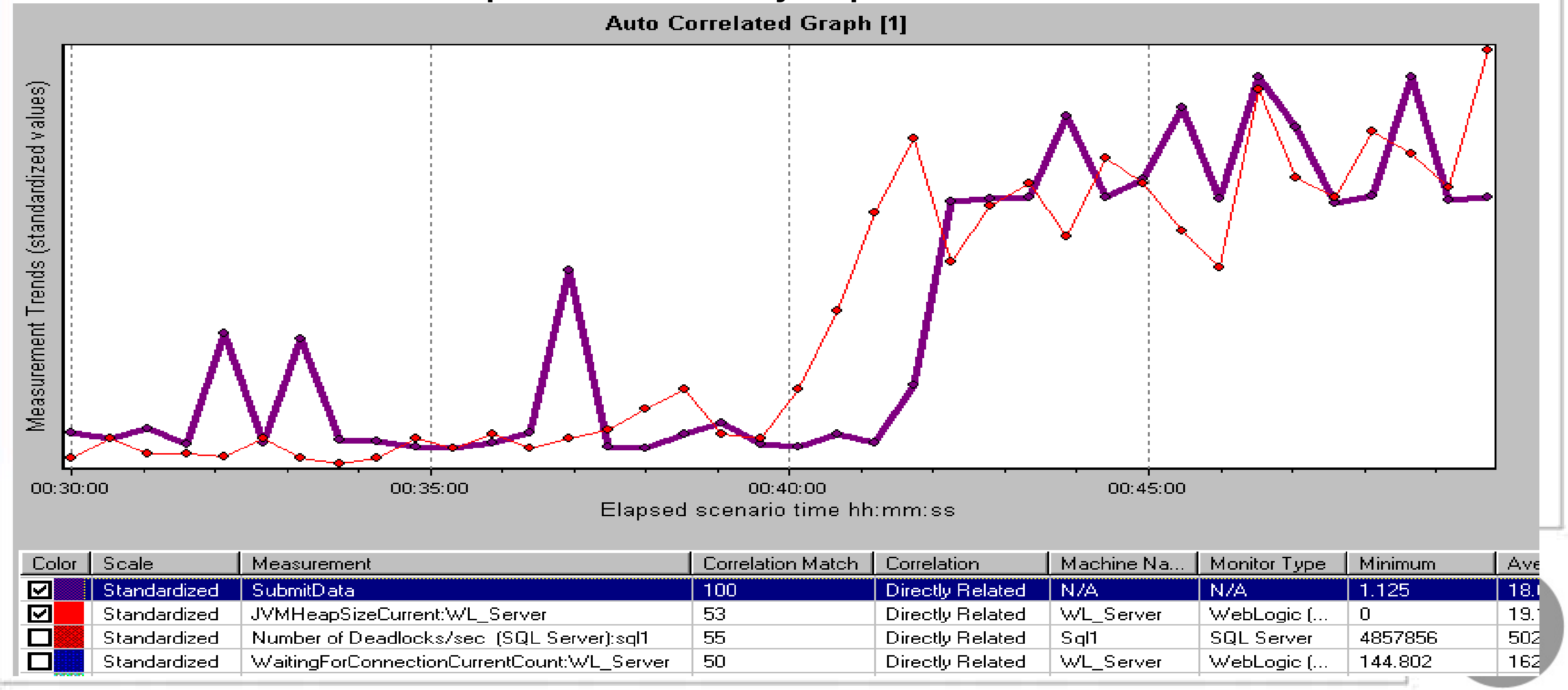
Monitor the Entire J2EE Stack



- Monitor all stack levels
- Monitor other system performance metrics
- Correlate system metrics with the end-user experience

Finding the Problem
Approach #1 - Divide-and-Concur

- Measure the bottom line
- Pinpoint the problematic Silo
  - Use automatic tools to detect prime suspects
  - for example: Auto Correlate in LoadRunner
- And then drilldown into this silo
  - Look into the details of each silo and all of their layers
  - In each step find the major performance bottleneck


Auto Correlated Graph [1]
Measurement Trends (standardized values)
00:30:00
00:35:00
00:40:00
00:45:00
Elapsed scenario time hh:mm:ss

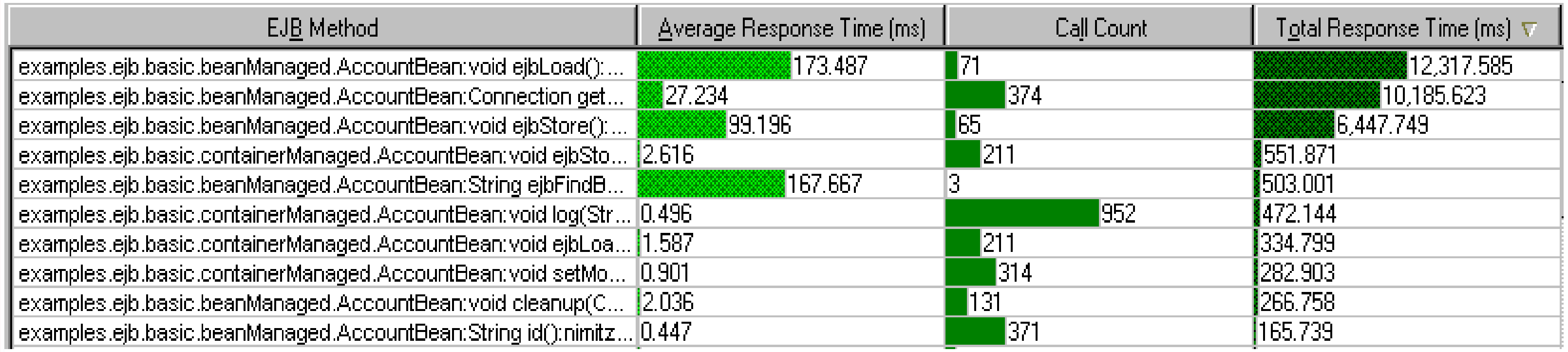
Color	Scale	Measurement	Correlation Match	Correlation	Machine Na...	Monitor Type	Minimum	Ave
☑ ▨	Standardized	SubmitData	100	Directly Related	N/A	N/A	1.125	18.
☑ 🟥	Standardized	JVMHeapSizeCurrent:WL_Server	53	Directly Related	WL_Server	WebLogic (...	0	19.
□ 🟥	Standardized	Number of Deadlocks/sec (SQL Server):sql1	55	Directly Related	Sql1	SQL Server	4857856	502
□ 🟦	Standardized	WaitingForConnectionCurrentCount:WL_Server	50	Directly Related	WL_Server	WebLogic (...	144.802	162

Drilling Down into J2EE

- There are various components to monitor:
  - JSPs/Servlets
  - EJBs
  - JNDI
  - JDBC/SQL-calls
  - JTA, JTS, JCA, JMS…
- For these components, we need to monitor:
  - Response time of methods
  - Number of times they are called
  - Number of exceptions thrown
  - Argument values
  - Total response time

Drilling Down into J2EE (cont.)

Important - “Total Response Time” !



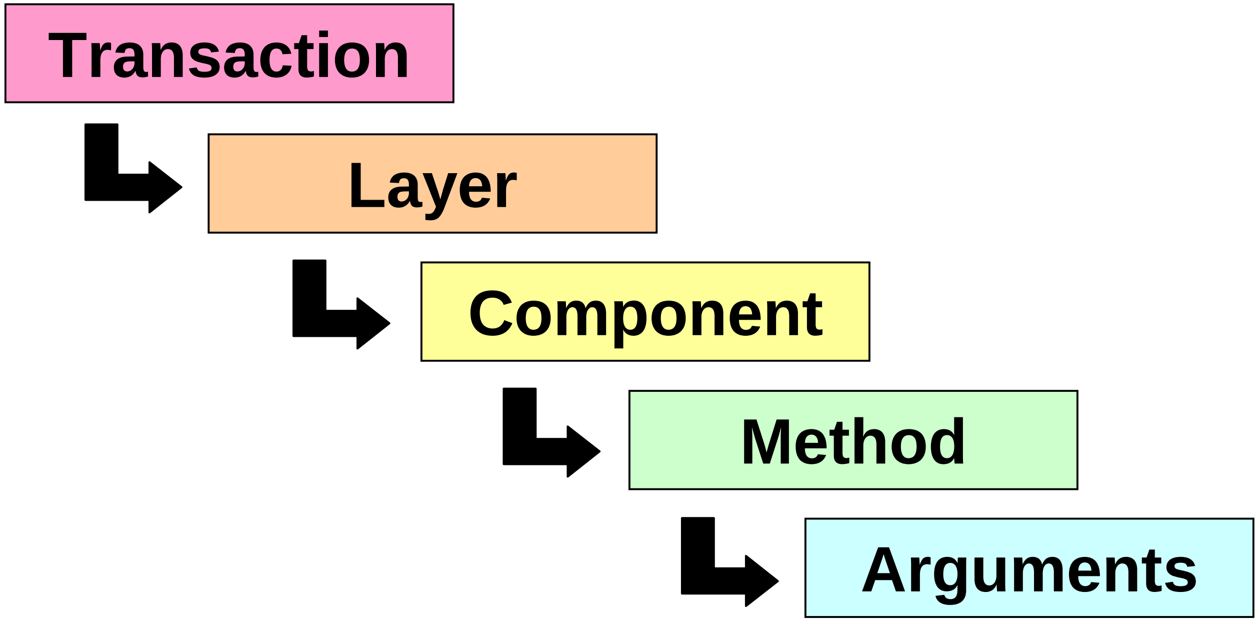
EJB Method	Average Response Time (ms)	Call Count	Total Response Time (ms) ▽
examples.ejb.basic.beanManaged.AccountBean:void ejbLoad():...	173.487	71	12,317.585
examples.ejb.basic.beanManaged.AccountBean:Connection get...	27.234	374	10,185.623
examples.ejb.basic.beanManaged.AccountBean:void ejbStore():...	99.196	65	6,447.749
examples.ejb.basic.containerManaged.AccountBean:void ejbSto...	2.616	211	551.871
examples.ejb.basic.beanManaged.AccountBean:String ejbFindB...	167.667	3	503.001
examples.ejb.basic.containerManaged.AccountBean:void log(Str...	0.496	952	472.144
examples.ejb.basic.containerManaged.AccountBean:void ejbLoa...	1.587	211	334.799
examples.ejb.basic.containerManaged.AccountBean:void setMo...	0.901	314	282.903
examples.ejb.basic.beanManaged.AccountBean:void cleanup(C...	2.036	131	266.758
examples.ejb.basic.beanManaged.AccountBean:String id():nimitz...	0.447	371	165.739

- Total Response Time =

  Avg. Response Time * number of calls
- Help to surface the real bottlenecks

Finding the Problem
Approach #2 – Top-Down

- Need to focus on specific “transactions”
- For “suspicious” ones, drill down:



Transaction

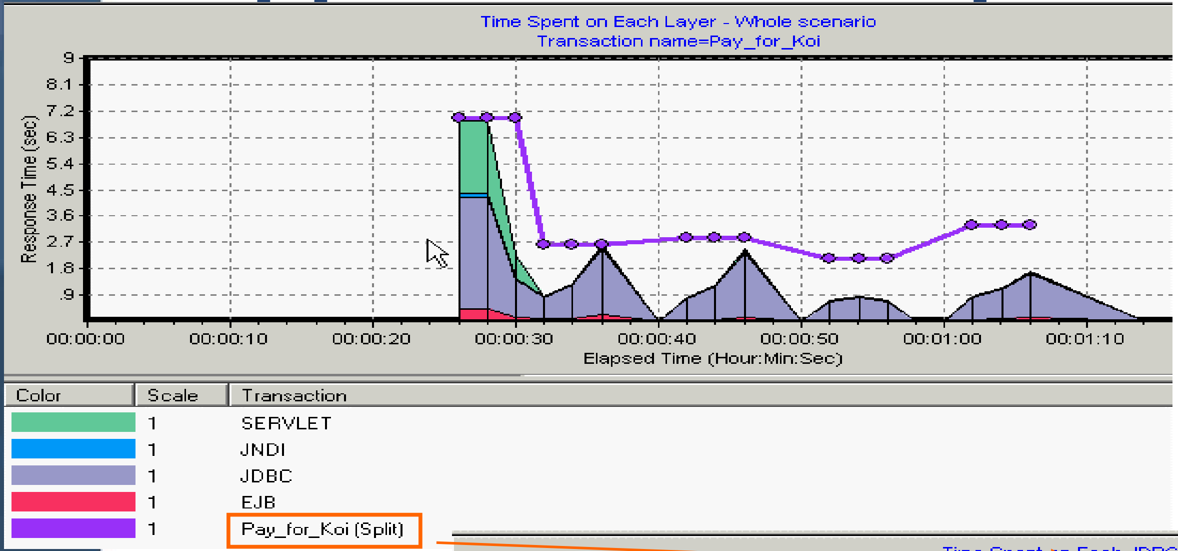
Layer

Component

Method

Arguments

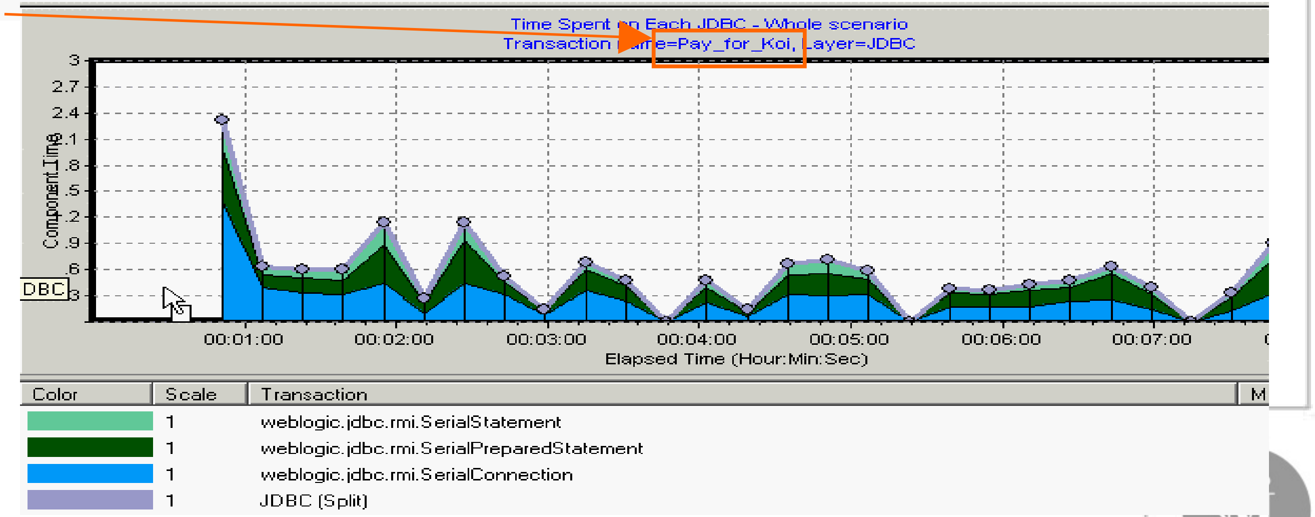
Finding the Problem
Approach #2 – Top-Down (Cont.)



Time Spent on Each Layer - Whole scenario
Transaction name=Pay_for_Koi

Color	Scale	Transaction
🟩	1	SERVLET
🟦	1	JNDI
🟪	1	JDBC
🟥	1	EJB
🟪	1	Pay_for_Koi (Split)

← Layer Level



Time Spent on Each JDBC - Whole scenario
Transaction name=Pay_for_Koi, Layer=JDBC

Color	Scale	Transaction	M
🟩	1	weblogic.jdbc.rmi.SerialStatement	
🟩	1	weblogic.jdbc.rmi.SerialPreparedStatement	
🟦	1	weblogic.jdbc.rmi.SerialConnection	
🟪	1	JDBC (Split)	

Class Level→

Summary

- Test your J2EE application early
- J2EE optimization is a reality - advance tools exist today
- Making sense out of what exists today is not a trivial task
- The ROI on these tools is huge

End

If You Only Remember One Thing…

J2EE performance problems exist. Detecting and solving them is necessary and a doable task!

End

Q&A