Unit and Integration Testing

Tools and Technologies for J2EE Applications

Stephen A. Stelting
Sun Services
Sun Microsystems, Inc.

Objectives and Payoff

- Objectives
  - Describe testing challenges for J2EE
  - Present open source testing technologies
  - Explain what to test in a J2EE application
- Payoff

  This session will provide a perspective of the technologies and approaches you can use for J2EE unit and integration testing

This talk is based around code demos, which are provided with the conference CD

When and Why Should You Test?

Two major schools of thought:

- "Test-first" methods or "test-driven development"
  - Create and run tests as you develop software
- Structured testing methodologies
  - Perform testing as an explicit phase of the software development cycle

Both methodologies advocate unit testing

Motivation: validate code, identify potential problems and bugs, check key assumptions, verify behavior under specified conditions

Common Types of Testing

- Black box tests
  - Tests external, observable code behavior
  - Includes user acceptance tests
- White box tests
  - Tests code "internals", functionality
  - Types of white box tests
    - Unit tests (testing a single component)
    - Integration tests (testing a bunch of components as a group)
- Regression tests: automated tests or test suites

Challenges of Testing in J2EE

- J2EE Components are:
  - Complex
  - Dependent on the framework
  - Tightly coupled with containers
  - Reliant on vendor-specific features (possibly)
- In addition, many components:
  - Store configuration information separately (DDs)
  - Interact with other J2EE components

Bottom line: it's hard to test J2EE code!

J2EE Testing Anti-Patterns

"Death by println": Use print or logging statements to test

Drawbacks:

- Not dynamic
- Code bloat becomes a problem
- Requires developers to sift through log files
- Hard to split out test code for production use

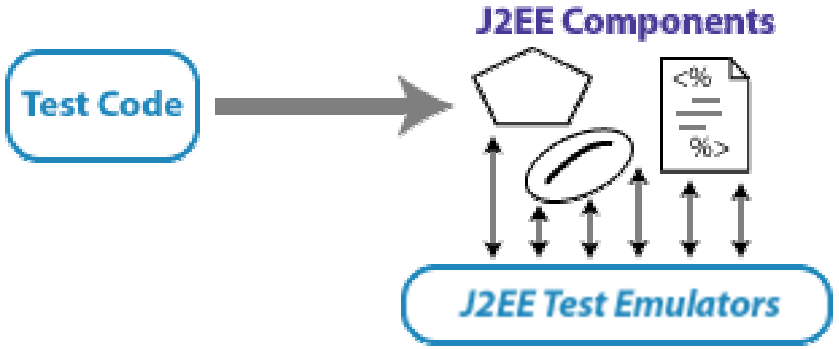
"Test-drive" Testing: Test the code by running it

Drawbacks:

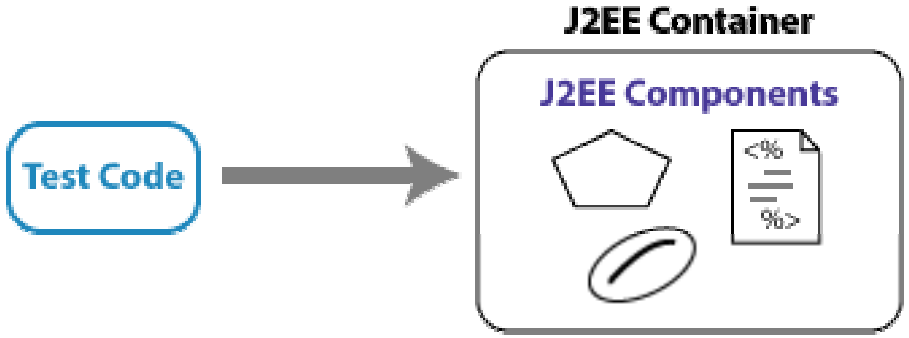
- Hard to isolate tests
- Limits developer testing to black-box techniques
- Not easily repeatable; causes problems with test automation, regression testing

Options for J2EE Testing

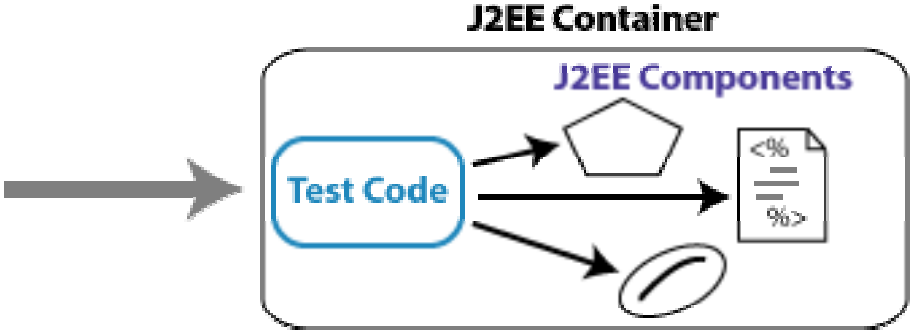
1) Test code, components outside the container

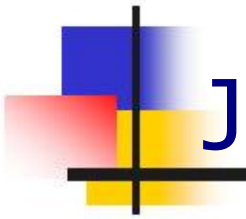


2) Test code outside, components inside the container



3) Test code, components inside the container



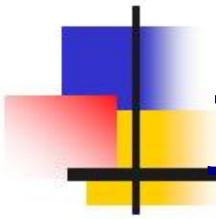


J2EE Test Techniques – Drawbacks

1. Test code, components outside the container
   - Requires test emulators
   - Doesn't provide container-specific services
2. Test code outside, components inside the container
   - Relies on communication channel to the container
   - Cannot use to unit test all J2EE components
   - Can be difficult to control test granularity
3. Test code, components inside the container
   - Requires some way to trigger the test code
   - Test code may depend on container services

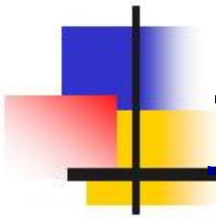
J2EE Testing Technologies

- Traditionally, J2EE testing frameworks have been provided by commercial vendors:
  - J2EE tool vendors (*e.g.* Sun, IBM, ...)
  - Testing specialists (*e.g.* Mercury Interactive)
- Three open source technologies are commonly used for J2EE unit testing:
  - jUnit
  - Mock objects
  - Cactus



jUnit

- jUnit: a generic unit test framework for Java
  - Available for most IDEs (Eclipse, NetBeans)
  - Used for regression testing, unit and integration tests
- Relevance to J2EE testing
  - jUnit can test within a container or outside
  - Not oriented towards testing J2EE components
- Mostly used for unit testing of Java helper classes (delegates, transfer objects, value objects)

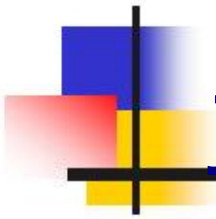


jUnit – Use in J2EE Testing

jUnit acts as a foundation for other frameworks that support J2EE unit testing

IDE support

- Eclipse: Supports auto-generation of jUnit test methods based around methods of a test class
- NetBeans: Support for template-based creation of base structure for tests and suites



jUnit – Code Structure

```
 1  import java.sql.*;
 2  import junit.framework.*;
 3  public class JUnitTest extends TestCase{
 4    private CustomerDAO dao;
 5    public void setUp() throws SQLException{
 6      dao = new CustomerDAO();
 7    }
 8    public void testCustomerDAOSelectByID()
 9      throws CustomerException{
10        Customer c = dao.selectByID(42);
11        TestCase.assertNotNull(c);
12    }
13  }
```

Mock Objects

Mock Objects: a generic unit testing framework

- Replaces domain code with emulators
- Many frameworks and projects are available, with various approaches to "mock" code

  EasyMock, MockEJB, Mock Objects, Mock Runner, Mock Creator, jMock, ... and many, many more!

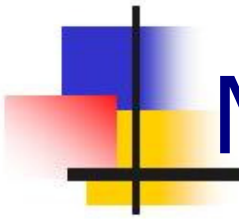
Mock Objects – Key Concepts

Developing unit tests for a component

1. Create component
2. Create mock objects
3. Call desired test behavior on component
4. Set expectations/check output

"Mocking" J2EE

- Mock objects usually test outside of a J2EE container
- Frameworks must "mock" the J2EE infrastructure
- Suitable for testing component functionality, with baseline assumptions about the infrastructure



Mock Object Testing in IDEs

- Neither Eclipse nor NetBeans have plug-in support for mock technologies
- However, most Mock frameworks are managed as one or more JAR files
- Include the JAR files in your IDE's CLASSPATH, and you can typically run unit tests using Mock objects

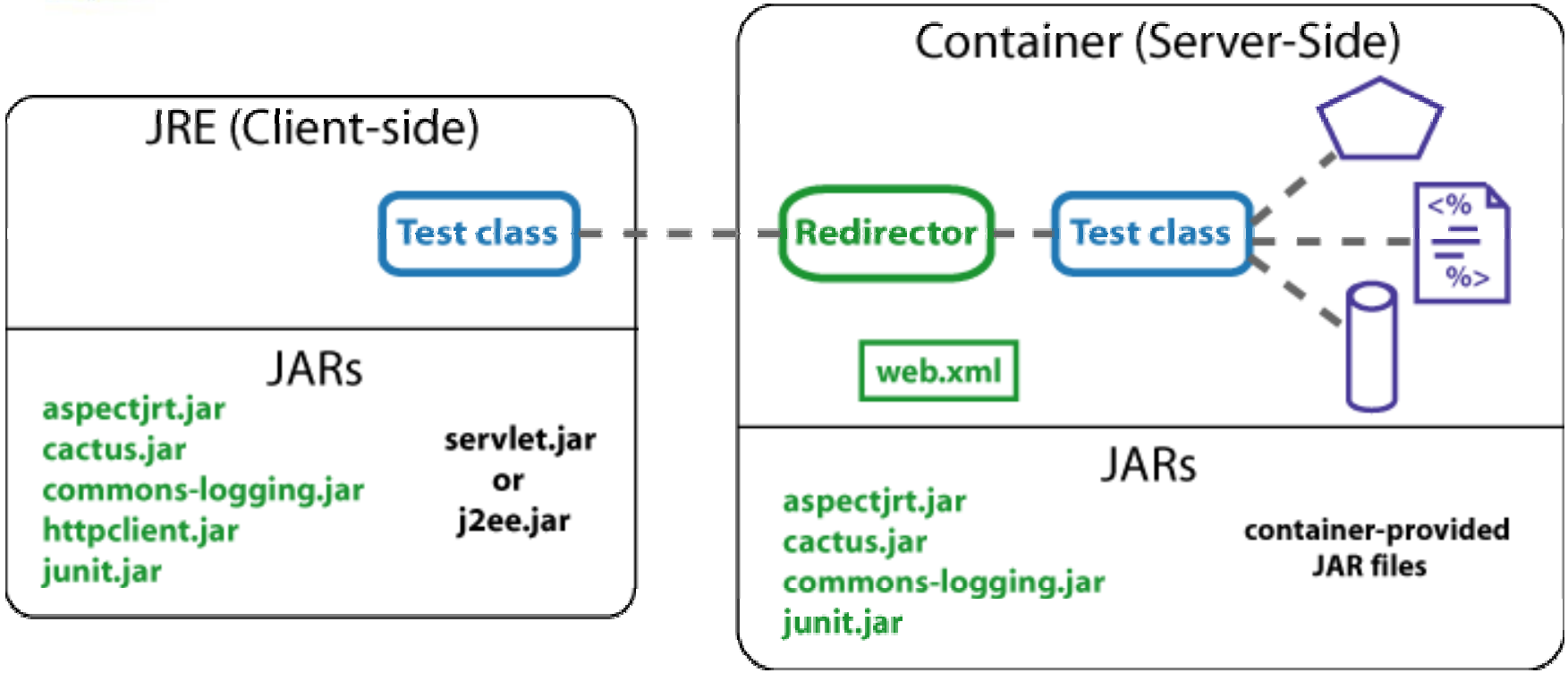
Mock Objects – Code Structure

```
 1 import java.io.*;
 2 import junit.framework.*;
 3 import com.mockrunner.servlet.*;
 4 public class MockRunnerTest
 5   extends ServletTestCaseAdapter{
 6     protected void setUp() throws Exception{
 7       super.setUp();
 8       createServlet(ControllerServlet.class);
 9     }
10     public void testServletRouting() throws Exception{
11       addRequestParameter("nav-action", "newCustomer");
12       doGet();
13       BufferedReader read = getOutputAsBufferedReader();
14       verifyOutputContains("Create New Customer");
15     }
16   }
```

Cactus

- Cactus: a J2EE unit testing framework
- Framework focuses on testing Web components: Filters, JSPs, Servlets
- Can also be used (indirectly) to test Tag Library code or EJBs
- Provides support for in-container testing

Cactus: Set-up and Testing



Optional config files: cactus.properites, log4j.properties

At a minimum, the client must be configured with the property cactus.contextURL (the URL of the Web server)

Cactus – Code Structure

```
 1 import javax.servlet.*;
 2 import junit.framework.*;
 3 import org.apache.cactus.*;
 4 public class CactusTest extends ServletTestCase{
 5   public void testControllerForward()
 6     throws ServletException{
 7       ControlServlet control = new ControlServlet();
 8       control.init(config);
 9       session.setAttribute("nav-action", "newCust");
10       String loc = control.getForwardURL(request);
11       ServletTestCase.assertEquals("/newCust.jsp", loc);
12   }
13 }
```

"Homegrown" Unit Tests

- If desired, you can create your own proxy components to test a J2EE system
- Common technologies by tier
  - Web: Filter or Servlet
  - EJB: Session Bean
- A proxy can act in various roles
  - Passive: data collector while a system runs
  - Active: client simulator, mimicking outside caller

Other Noteworthy Frameworks...

Although this presentation has focused on jUnit, Mock Objects and Cactus, there are a number of other useful open source test frameworks available:

- **DbUnit**
  - Supports unit testing of DBMS and allows you to import and export database data between tests
- **HttpUnit**
  - Unit test framework for Web sites and applications, including Java Web applications
- **htmlunit**
  - Unit test framework for HTML-based Web sites

What "Should" You Test in J2EE?

- Unit tests
  - Business methods
  - Lifecycle
- Integration tests
  - Components
    - Call chains
    - Composites (subsystems)
  - System response to infrastructure conditions
  - Response to (and recovery from) failure scenarios

Integration Tests

- Components
  - Flow of operations
  - Whole-part structure during lifetime
  - Appropriate delegation and cascading behavior
- Infrastructure conditions and failure scenarios
  - Service unavailability
  - Component unavailability
  - Communication latency
  - Resource "bottlenecks"

Other Testing Considerations

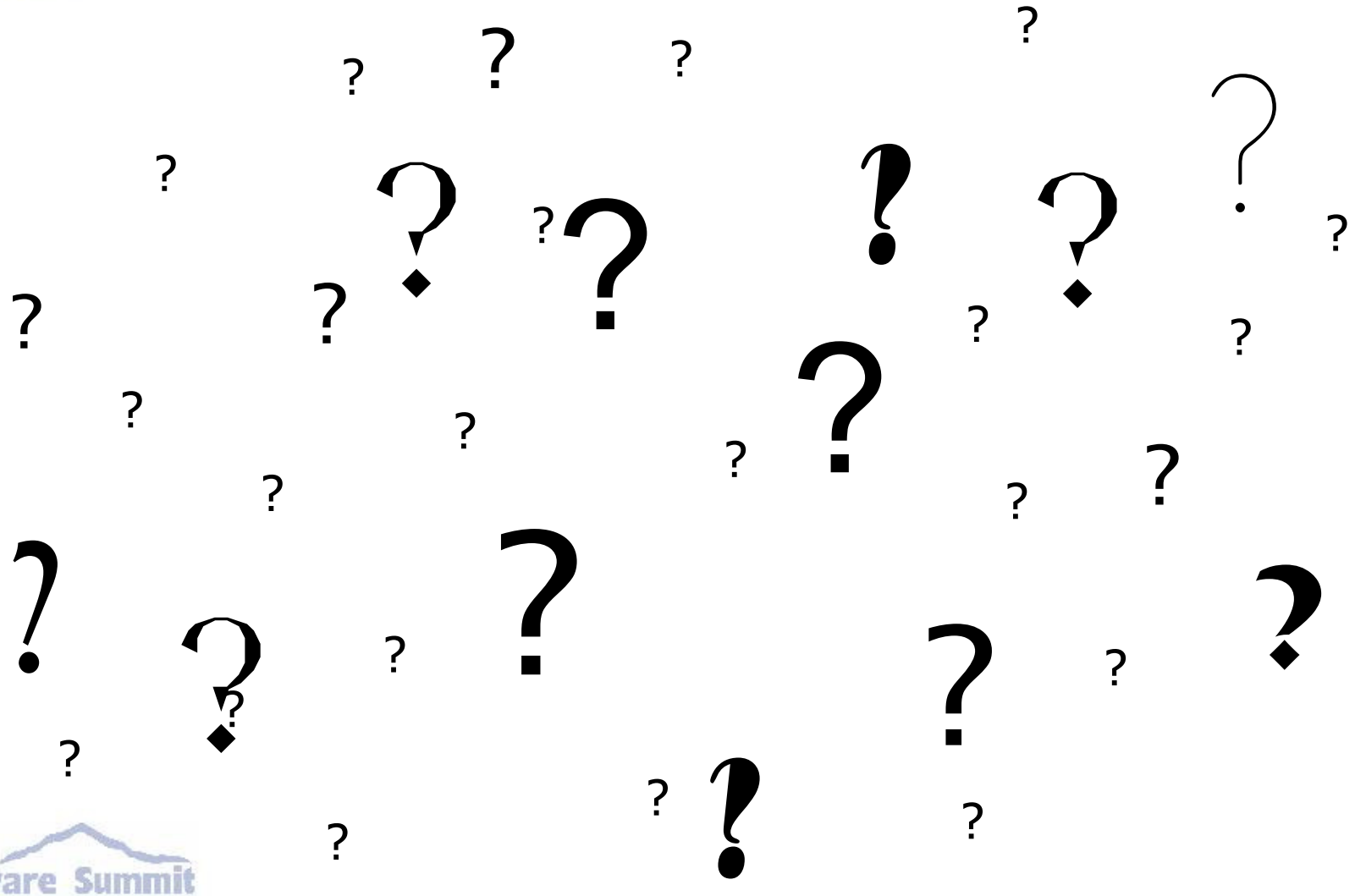
- Framework code
  - Struts
  - JavaServer Faces
- Architectural testing
  - Performance
  - Stress testing

Summary

In this session, we have:

- Described the foundation principles of unit testing for J2EE applications
- Presented the open-source technologies commonly used for unit testing
- Demonstrated how to set up and run J2EE unit tests using Eclipse and NetBeans

Questions



URL References

Technologies

- jUnit: http://www.junit.org/
- Mock objects: http://www.mockobjects.com/
- Easy Mock: http://www.easymock.org/
- Cactus: http://jakarta.apache.org/cactus
- HttpUnit: http://httpunit.sourceforge.net/
- dbUnit: http://dbunit.sourceforge.net/
- htmlunit: http://htmlunit.sourceforge.net/

IDE Tools

- Eclipse: http://www.eclipse.org/
- NetBeans: http://www.netbeans.org/