CONTINUOUS REGRESSION TESTING FOR JAVA™ EE APPS: CHANGE CODE WITHOUT FEAR

Matt Love, Software Development Manager

TS-7669

Java

Sun microsystems

Learn how to catch Java™ Platform, Enterprise Edition (Java EE) integration issues using continuous regression tests

GOAL

Agenda

- Problem: traditional approaches for testing Java EE platform
- Solution: continuous regression testing for Java EE platform
- To JUnit and beyond: extending the regression test suite
  - Step 1: Identify the goals
  - Step 2: Define the scope
  - Step 3: Select a framework
  - Step 4: Write tests within the corresponding test framework
- Example: Integration testing out-of-container with Spring
- Automated infrastructure for continuous regression testing

Traditional Java EE Platform Test Approach

Unit and system tests at both extremes miss the middle

- JUnit Tests
  - Test plain object code (POJOs)
  - Test each unit in isolation
- Benefits
  - Low maintenance overhead
    - Independent of other tests/units
    - Easy to add or modify tests
  - Quick to run tests
- Drawbacks
  - Integration errors are missed
  - No framework dependent code

- System Tests
  - Scripted end-to-end tests
  - Exercise the whole application
- Benefits
  - Detect integration errors
  - Test real use cases
- Drawbacks
  - Slow to run
    - Prohibitive for the developer
  - High maintenance effort
    - Sensitive to changes in any layer

Unit Tests and System Tests
Still not enough

- Ordinary JUnit will not find integration problems
  - Framework configuration files are not used in JUnit tests
    - web.xml, struts-config.xml, applicationContext.xml
  - Database operations are not verified
- System tests are difficult to run on developer workstations
  - External dependencies require extra setup work
    - Application Server
    - Database
    - Test scripts for the presentation layer
  - System testing is slow for new code changes
    - Recompile & deploy all modules
    - Wait for all layers in Java EE platform system to initialize
- Developers postpone testing Java EE platform problems

Agenda

- Problem: traditional approaches for testing Java EE platform
- Solution: continuous regression testing for Java EE platform
- To JUnit and beyond: extending the regression test suite
  - Step 1: Identify the goals
  - Step 2: Define the scope
  - Step 3: Select a framework
  - Step 4: Write tests within the corresponding test framework
- Example: Integration testing out-of-container with Spring
- Automated infrastructure for continuous regression

What is Continuous Regression Testing?

Java EE Platform Integration Testing outside Container

- Not a replacement for Unit Tests or System Tests
  - Hybrid approach fills the gap
  - Run alongside other tests in the IDE or build server
- Detect more problems than JUnit
  - Test interactions between two layers in the Java EE platform
  - Catch regressions with respect to framework configuration files
- Test sooner than scripted system tests
  - Before committing code changes to source control
  - Before other modules or layers have been developed
  - Execute nightly or as part of continuous integration

Why Bother?

Extend best practices to Java EE platform artifacts

- Agile Development
  - Short development iterations
  - Early feedback on the effects of changes
- Test During Development (the other TDD)
  - Find and fix defects during before committing to source control
  - Enhance productivity through shorter defect cycles
- Reduce demand on QA
  - No wasted time on module integration errors
  - Focus on validating end-user scenarios
- More flexible
  - Fewer module-module and module-framework dependencies
    - Change one module without affecting other module design or tests
    - Swap Java EE platform frameworks while preserving some tests

Agenda

- Problem: traditional approaches for testing Java EE platform
- Solution: continuous regression testing for Java EE platform
- To JUnit and beyond: extending the regression test suite
  - Step 1: Identify the goals
  - Step 2: Define the scope
  - Step 3: Select a framework
  - Step 4: Write tests within the corresponding test framework
- Example: Integration testing out-of-container with Spring
- Automated infrastructure for continuous regression

Step 1: Identify the Goals

Tests exist for a reason more important than coverage

- Functionality verification – "does this work?"
  - Correlate tested behavior to specification
  - Grow the regression suite for continued integrity of key functionality
- Bug-finding
  - Vet unexpected behavior to find bugs early
  - Unexpected scenarios can be generated automatically
  - Add to the regression suite to ensure problems don't recur
- TDD
  - Write tests before code, or…
  - Write tests for every reported problem
  - Verify the test by watching it fail before the problem is fixed

Step 2: Define the Scope

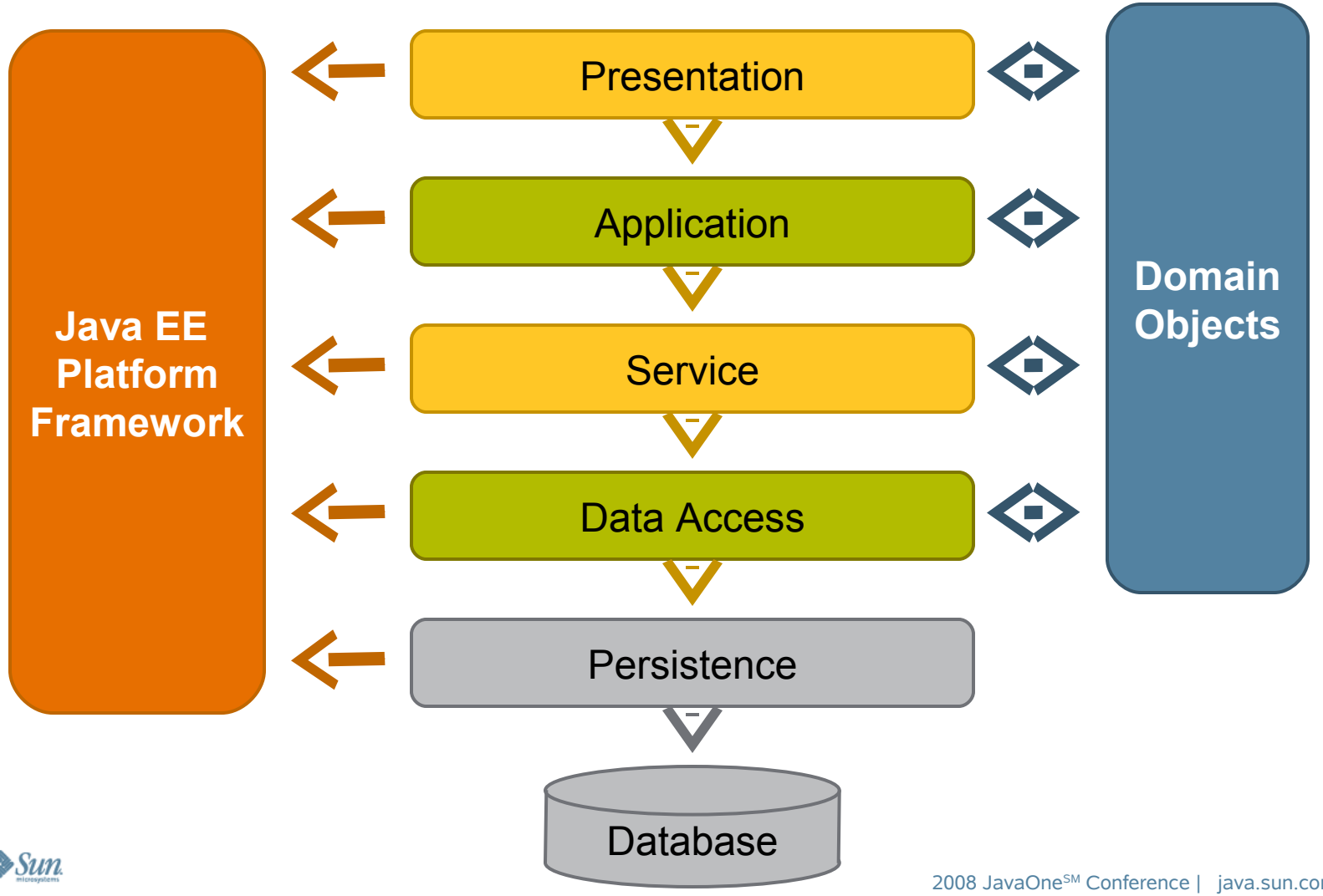
With the test goal in mind

- Scope reduction
  - All requirements or problems can be tested at the system level
  - Some layers can be removed while preserving the same test goal
  - Identify the core logic and layer interaction for each case

- Unit tests are like onions and ogres: they all have layers
  - Code separation layers
    - Classes, packages, bundles
  - Logical separation layers
    - Work units, independent functionality, unrelated requirements
  - Framework dependency layers
    - Java EE platform imposed levels of abstraction

Java EE Platform Application Architecture

Typical Dependency on Layers and Framework



Scope of Testing

Component	Test Strategy	Rationale
Plain object code	JUnit	Test each method to capture current behavior and intra-method contracts
Framework dependent code	Mock Objects	Spoof framework API using mock objects to exercise code in isolation
Framework configuration	Framework integration tests	Use the testing pattern provided by framework for best results
Persistence	DBUnit / ORMUnit	Put database in a known state for testing data access and modification
XML	XMLUnit	Compare XML data against control and report logical data differences
System	Scenario scripting	Full end-to-end test for final validation or repeated for load testing

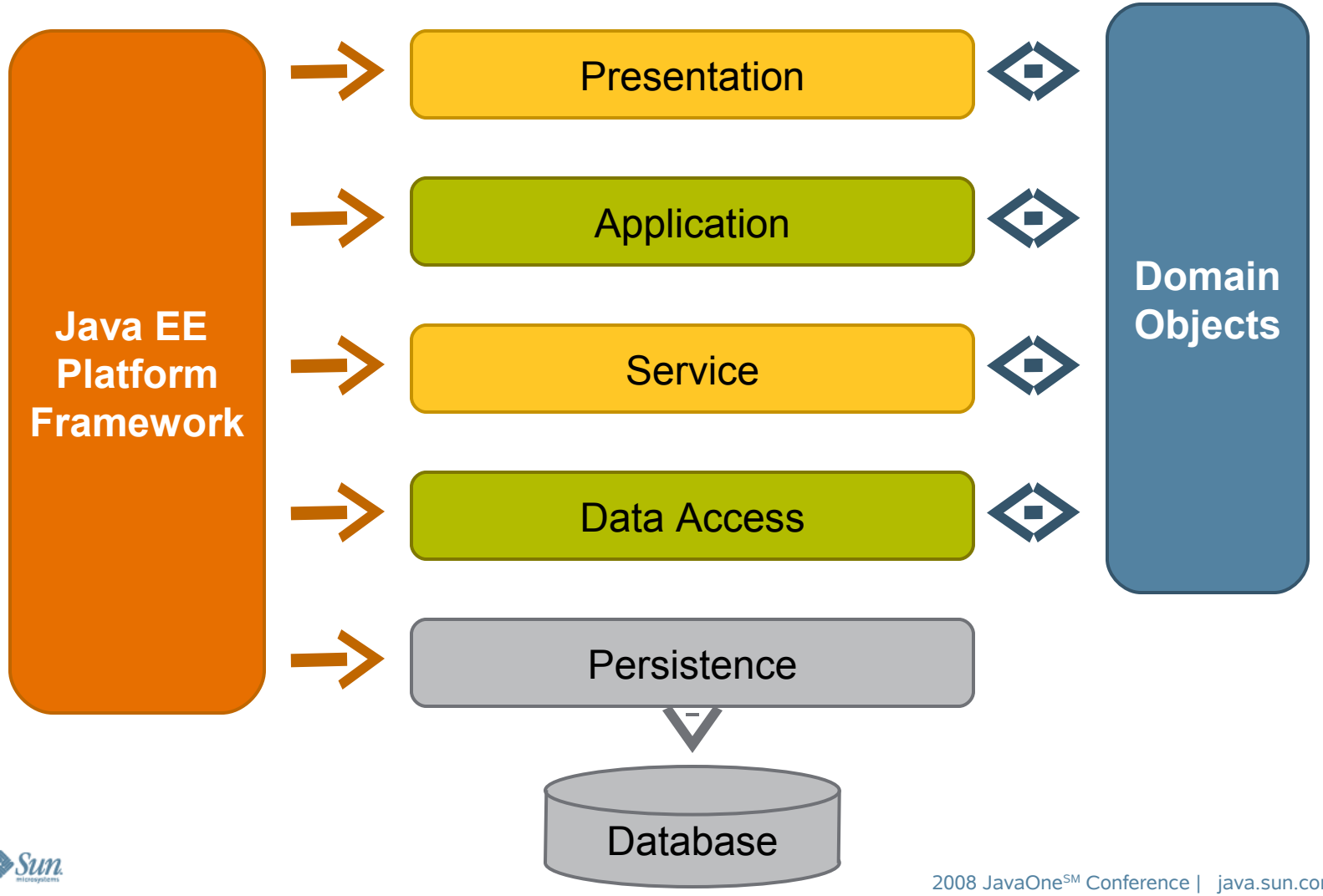
Step 3: Select a Framework

To best support agile projects and integration testing

- Common framework abilities in the Java EE platform
  - Manage all technology layers in the container
  - Simplify the raw Java EE platform interface
  - Configuration without code
    - XML
    - Properties
    - Manifest
- Desired framework qualities
  - Inversion of Control (IoC)
    - Also known as Dependency Injection (DI)
    - Code is written without explicit dependency on framework or layers
    - Framework wires implementations and dependencies together
  - Layers of Abstraction
    - Data access
    - Transaction management
  - Specialized harness for testing outside the container

Java EE Platform Application Architecture

Dependency Injection from Framework



Struts

- Code design
  - Invasive
    - Must extend classes in org.apache.struts.action
    - Little Inversion of Control due to direct calls to Struts API
  - Data access and persistence left open; not regulated by Struts
- Testing
  - Helpers available for testing Struts Action Forms
  - In container
    - Tests extend servletunit.struts.CactusStrutsTestCase
    - Inherit from org.apache.cactus.ServletTestCase
  - Outside container
    - Tests extend servletunit.struts.MockStrutsTestCase
    - Inherit from junit.framework.TestCase

Spring

- Code design
  - Non-invasive
    - Few calls to Spring API
    - Setter injection of DAO into bean classes
    - Dependencies specified in applicationContext.xml
  - Data access objects and persistence connected through Spring
- Testing
  - Spring test framework to create bean and DAO from configuration
  - Outside container testing from service to database interaction
    - Tests extend AbstractTransactionalDataSourceSpringContextTests
    - Database changes from each test are reverted
  - Tests make simple calls to POJO domain objects

Aspect Oriented Programming
Enterprise JavaBeans™ 3 (EJB™ 3) Architecture

- Code design
  - Inversion of Control
    - Annotations to configure project layers
    - Aspects instrumented in the container
- Testing
  - Project code testable out of container as POJOs without aspects
  - Ejb3Unit from sourceforge for testing the code outside of container
    - In memory Java Naming and Directory Interface (J.N.D.I.) server
    - Bean creation with dependencies injected

Eclipse Plug-In Development

The same rules apply beyond the Java EE platform

- Code design
  - Dependency injection through plug-ins and extension points
- Testing
  - JUnit plug-in test runner
    - Spawns new Eclipse runtime workspace for JUnit tests
    - Equivalent to in-container testing
    - Code entry points to bypass slow modules or UI
  - Use traditional JUnit to test domain objects
  - Mock objects to simulate Eclipse API
    - Generic mock frameworks: EasyMock, jMock
    - Unfortunately mocks for Eclipse API are not predefined

Step 4: Write framework tests

- Use test pattern recommended for the specific framework
  - Extend the base class provided by the test framework
    - JUnit inheritance
    - Framework test API available to test method scope
  - Initialize the test environment
    - Specify configuration files
    - Redefine system properties
  - Restore the test environment when finished
    - Roll back persistent changes or side effects
- Design tests for maintainability
  - Identify which use case is being tested
  - Convey as much information in failure messages as possible
  - Negate all side effects so tests are independent from each other

Agenda

- Problem: traditional approaches for testing Java EE platform
- Solution: continuous regression testing for Java EE platform
- To JUnit and beyond: extending the regression test suite
  - Step 1: Identify the goals
  - Step 2: Define the scope
  - Step 3: Select a framework
  - Step 4: Write tests within the corresponding test framework
- Example: Integration testing out-of-container with Spring
- Automated infrastructure for continuous regression

Example: Spring Test Framework
iBATIS JPetStore 5

- Project components
  - Spring dependency injection
  - iBATIS persistence layer
  - HSQLDB in memory database
  - Spring or Struts web tier (presentation + application layer)
- Spring Test Framework
  - Exercise service logic, DAO and DB out-of-container
  - Roll back DB transactions after each test

Spring bean configuration

JPetStore business object in applicationContext.xml

```
<bean id="petStore" parent="baseTransactionProxy">
<property name="target">
<bean
class="org.springframework.samples.jpetstore.domain.logic.
PetStoreImpl">
<property name="accountDao" ref="accountDao"/>
<property name="categoryDao" ref="categoryDao"/>
<property name="productDao" ref="productDao"/>
<property name="itemDao" ref="itemDao"/>
<property name="orderDao" ref="orderDao"/>
</bean>
</property>
</bean>
```

Spring bean implementation class

JPetStore business object does not reference Spring

```
public class PetStoreImpl
    implements PetStoreFacade, OrderService
{
  private AccountDao accountDao;
  private ItemDao itemDao;
  private OrderDao orderDao;
  public void setAccountDao(AccountDao accountDao) {
    this.accountDao = accountDao;
  }
  public void setItemDao(ItemDao itemDao) {
    this.itemDao = itemDao;
  }
  public void setOrderDao(OrderDao orderDao) {
    this.orderDao = orderDao;
  }
}
```

Spring Test Class Template

```
public class PetStoreSpringTransactionalDataSourceTest
extends AbstractTransactionalDataSourceSpringContextTests
{
  @Override
  protected String[] getConfigLocations() {
    return new String[] {
      "/WEB-INF/applicationContext.xml",
      "/WEB-INF/dataAccessContext-local.xml" };
  }
}
```

JPetStore Integration Test Case

Spring Dependency Injection Test Framework

```
public void testPetStoreOrderProcessing() {
  PetStoreFacade petStore = (PetStoreFacade)
    applicationContext.getBean("petStore");
  Account account = petStore.getAccount("j2ee");
  Item spottedKoi = petStore.getItem("EST-4");
  Item persianCat = petStore.getItem("EST-16");
  Cart cart = new Cart();
  cart.addItem(spottedKoi, true);
  cart.addItem(persianCat, true);
  Order order = new Order();
  order.initOrder(account, cart);
  petStore.insertOrder(order);
  Order dbOrder = petStore.getOrder(order.getOrderId());
  assertEquals(spottedKoi.getListPrice() +
    persianCat.getListPrice(), dbOrder.getTotalPrice());
}
```

Log4j Output

TransactionalDataSourceSpringContextTest

```
INFO: Loading config for: /WEB-INF/applicationContext.xml
INFO: JDK 1.4+ collections available
INFO: Commons Collections 3.x available
INFO: Loading XML bean definitions from class path
resource [WEB-INF/applicationContext.xml]
INFO: Defining beans [propertyConfigurer,
  accountValidator, orderValidator, baseTransactionProxy,
  petStore, dataSource, transactionManager, sqlMapClient,
  accountDao, categoryDao, productDao, itemDao, orderDao,
  sequenceDao]; root of BeanFactory hierarchy
INFO: 14 beans defined in application context
INFO: Loading properties file from class path resource
[WEB-INF/jdbc.properties]
INFO: JDBC 3.0 Savepoint class is available
INFO: Began transaction (1): DataSourceTransactionManager
INFO: Rolled back transaction after test execution
```

Agenda

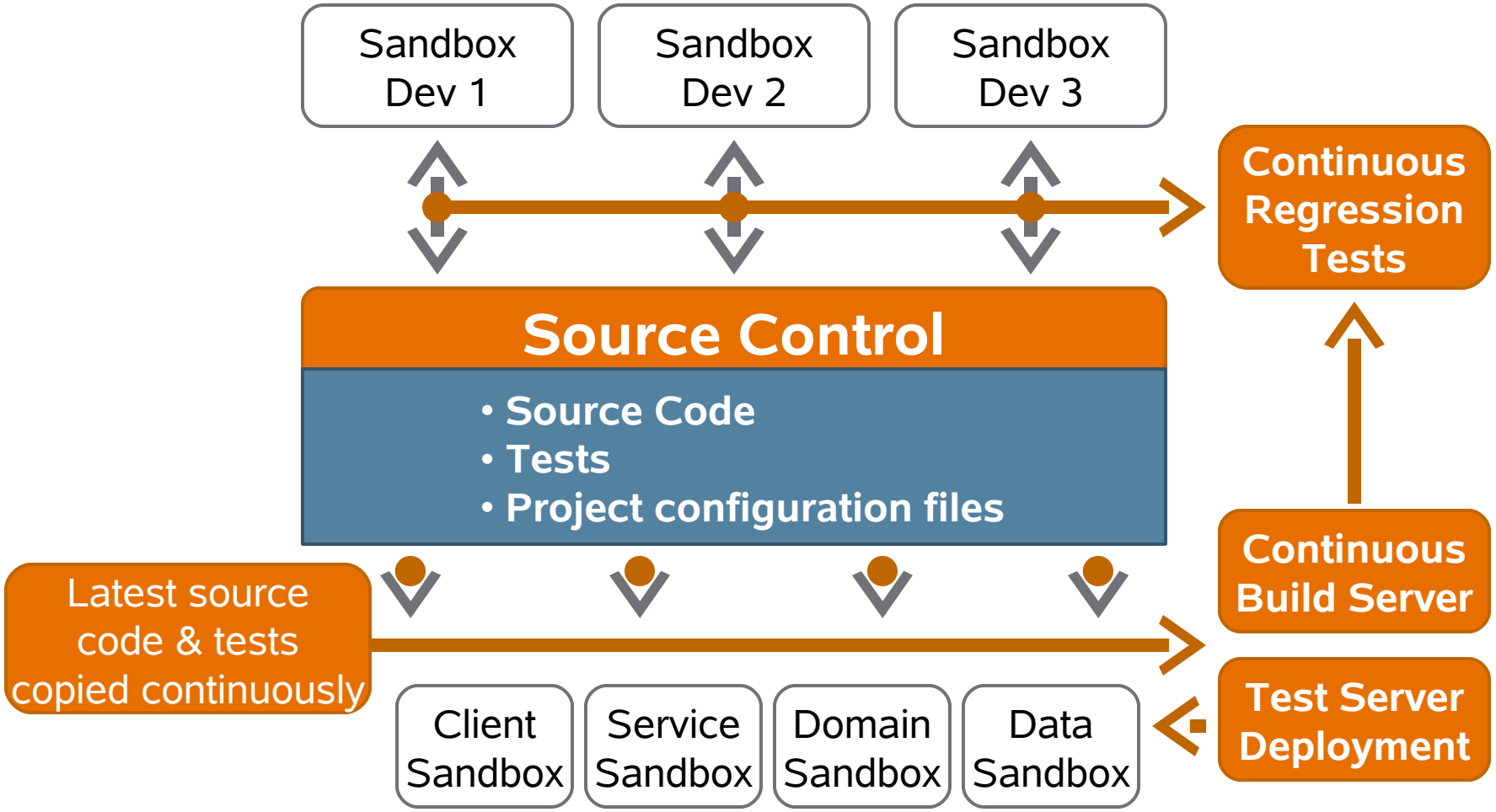
- Problem: traditional approaches for testing Java EE platform
- Solution: continuous regression testing for Java EE platform
- To JUnit and beyond: extending the regression test suite
  - Step 1: Identify the goals
  - Step 2: Define the scope
  - Step 3: Select a framework
  - Step 4: Write tests within the corresponding test framework
- Example: Integration testing out-of-container with Spring
- Automated infrastructure for continuous regression

Establishing an Automated Infrastructure

- Why?
  - Tests are always run and verified – never forgotten
  - Defects are uncovered sooner – before QA testing
    - Modules that change can be rebuilt and tested immediately
  - Productivity improvements
    - Offload time and effort to check tests to dedicated servers
    - Code changes with confidence that regressions will be caught
    - Regression defects are fixed faster on fresh changes
    - New tests are added quickly by extending other tests
- How?
  - Ant – schedule routine builds and tests
  - CruiseControl – web dashboard for build schedule and test results
  - Maven – standard project structure
  - Continuum – source control xml-rpc commit trigger & change log

Java EE Platform Development Model

with automation for continuous testing



Summary

- Continuous regression testing provides a safety net to change Java EE platform code without fear
  - Fear of breaking container framework configuration
  - Fear of breaking business logic
  - Fear of breaking database transactions
- Execute integration tests nightly (at least)
  - Test for (and fix) defects as soon as possible
  - Run integration tests before committing code when in doubt
- Every layer counts
  - JUnit tests
  - Mock framework tests
  - System tests
- Establish an automated infrastructure to drive the process

For More Information

- Published Articles
  - Love, Matt. “Extend Beyond JUnit.” Software Test and Performance August 2007: 26-30
  - daVeiga, Nada. “Change Code Without Fear.” Dr. Dobb’s Journal February 2008
- Books
  - Huizinga, Dorota and Adam Kolawa. Automated Defect Prevention. New Jersey: John Wiley & Sons, Inc. 2007
- On the web
  - http://www.springframework.org/
  - http://struts.apache.org/
  - http://continuum.apache.org/
  - http://www.parasoft.com/
  - http://ejb3unit.sourceforge.net/

THANK YOU

Matt Love, Software Development Manager

TS-7669