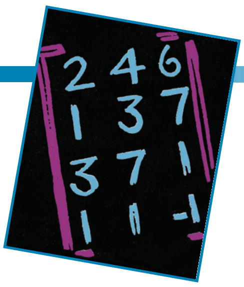
1-2007

A Hike through a Post-EJB J2EE Web Application Architecture, Part III,

Konstantin Laufer

George K. Thiruvathukal

Benjamin Gonzalez



Editors: George K. Thiruvathukal, gkt@cs.luc.edu
Konstantin Läufer, laufer@cs.luc.edu

A Hike through a Post-EJB J2EE Web Application Architecture, Part III

By Konstantin Läufer, George K. Thiruvathukal, and Benjamín González

By incorporating automated component, integration, and acceptance testing into the various tiers of a lightweight Java 2 Enterprise Edition (J2EE) Web application architecture, developers can shorten the development cycle and increase the quality of their work.

In this third article in a series about Web application development—the first[1] focused on the upper tiers, and the second[2] focused on the middle and lower tiers—we retrofit the entire architecture with support for automated testing.[3]

We continue here with our running example—linear regression over a persistent set of points—and our familiar blueprint of a typical post-Enterprise Java Beans (EJB) Java 2 Enterprise Edition (J2EE) Web application architecture (see Figure 1). In past hikes, we first worked our way up through the view and controller layers in the user interface tier, and then made our way down through the application and data model layers in the middle tier until we reached the persistence tier at the bottom. We looked at typical technologies for each layer and used lightweight containers to painlessly connect the various pieces.

In this article, we add support for automated component, integration, and acceptance testing. We'll test isolated components in each tier and then do some integration testing with other components. Once we get all the way to the top, we'll perform acceptance testing on the entire Web application as the user would see it. Our objective is to gain confidence in the correctness of all application-specific components, their integration with each other, and their integration with components from various frameworks.

If you still need convincing about the important benefits of test-driven development (TDD), please read Kent Beck and Erich Gamma's "Test Infected: Programmers Love Writing Tests" (http://members.pingnet.ch/gamma/junit.htm). Otherwise, let's get started.

To follow a proper TDD approach, in which coding and testing go hand in hand, we should build the testing instrumentation into our example code from the start. However, to keep the presentation clear and simple, we kept testing out of the earlier installments of this series and made it the main focus of this one. To make the article as self-contained as possible, we repeat some of the examples from the earlier articles where appropriate. The example code is also available at http://snapshots.cs.luc.edu/338/LinearRegression and :pserver:anonymous@cvs.cs.luc.edu:/root/laufer/338.

The Dependency Graph: Figuring out the Best Route

Naturally, the first question is where to start, so let's turn to the equivalent of a trail map for this hike: an inter-package dependency graph, which indicates compile-time dependencies among the application's various packages. In a Java Web application, we typically have at least one package per tier. In our running example, these packages include

- `points.web`, the application-specific components of the user interface tier's controller layer (the actions behind the application's dynamic behavior);
- `points.biz`, the application model in the middle tier's upper half (the service object that exposes linear regression functionality to the user interface tier); and
- `points.hib`, the data model in the middle tier's lower half (the data access object [DAO] manages application data by mapping it to persistent storage).

Ideally, each tier in a tiered architecture depends only on the next-lower level, so the dependency graph should resemble a linear chain. However, two issues arise on closer inspection: first, where should the `Point` interface go? If we put it in the data tier, the Web tier depends on the data tier even though it should depend only on the business tier. But if we put it in the business tier, the data tier depends on the business tier, and we'll have an undesirable cycle in the dependency graph. Second, what if we want to switch data model implementations without changing data model interfaces? Fortunately, we can address both issues by introducing two more packages:

- `points.common`, the interfaces used across various tiers (in this case, only the `Point` interface); and
- `points.dao`, the data model's implementation-independent portion (that is, the DAO interface). By contrast, the data model's Hibernate-specific (www.hibernate.org) portions stay in the `points.hib` package.

In general, we should attempt to make the inter-package dependency graph acyclic by moving types between packages or splitting up packages as needed.

Using the Metrics plug-in for Eclipse (http://metrics.sourceforge.net), we can generate inter-package dependency graphs for Java applications automatically, among other useful things. Figure 2 shows the dependency graph for our linear regression example. The wider part of the connection represents the base of the arrow, and the pointed part represents the tip. The tiny boxes with numbers in them indicate additional dependencies on external packages. The portion of the dependency graph corresponding to the Web, business, and implementation-independent data tiers is a linear chain pointing down as expected. The implementation-specific data tier points up on the implementation-independent data tier as expected because it implements that tier's interfaces. From a functionality viewpoint, of course, the business tier also depends on the implementation-specific data tier, so a cycle here doesn't show up in the inter-package dependency graph because the Spring framework (www.springframework.org) manages this dependency at runtime.

The next step is to identify all the sinks in the dependency graph—that is, those packages that don't depend on any other packages. Because the graph is non-empty and acyclic, we're guaranteed at least one sink. In this case, it's the `points.common` package, which contains the `Point` interface. Because we can test only concrete classes, we have to remove the node corresponding to this package and try again. This leaves us with the `points.dao` package as the next sink, which again contains only the `PointsDAO` interface, so we have to remove it, too. As mentioned earlier, even though the business tier doesn't seem to depend explicitly on the Hibernate tier, it has an implicit runtime dependency, so we proceed with component and integration testing from the lowest to the highest tier, in reverse order of dependency.

Testing on Specific Tiers

Let's look at component and integration testing in the specific tiers, using additional technology beyond straight JUnit (www.junit.org) where appropriate. Following common practice, we put the tests for each tier in the corresponding

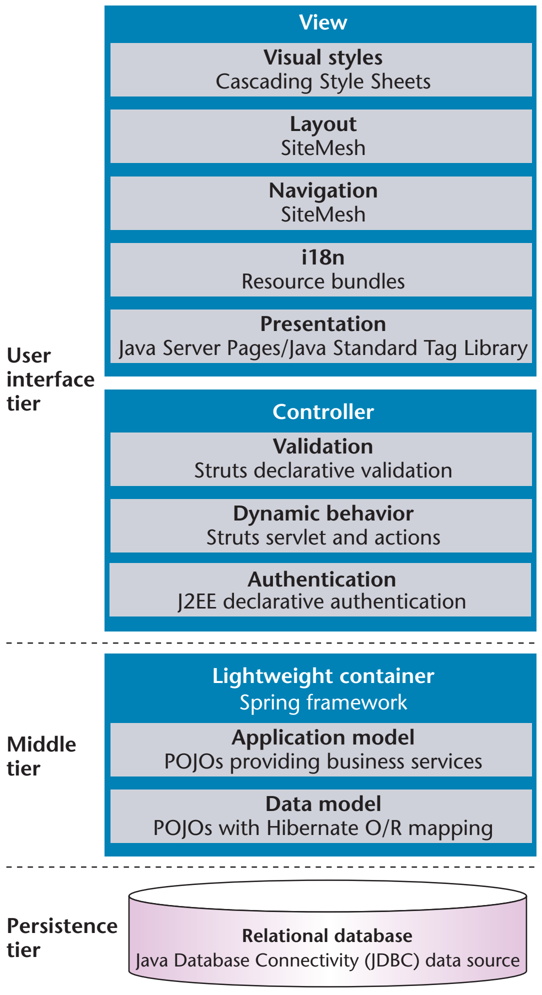


Figure 1. Architectural blueprint. The post-Enterprise Java Beans (EJB) Java 2 Enterprise Edition (J2EE) Web application architecture shows the typical tiers to include in a comprehensive testing strategy.

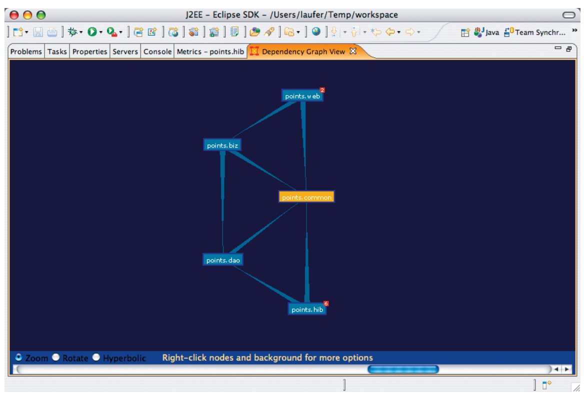


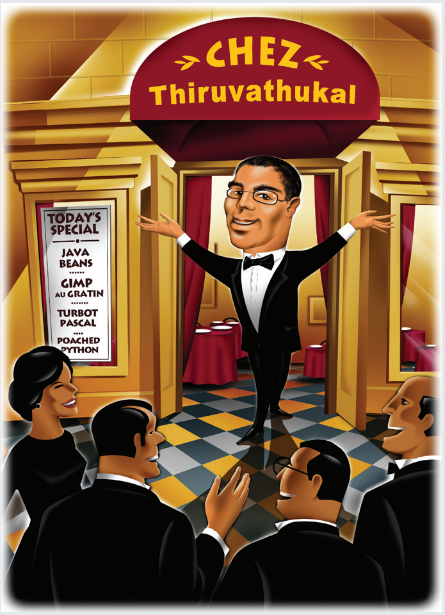
Figure 2. Inter-package dependency graph for the linear regression example. Each tier depends only on the next lower tier and common interfaces; only the lowest tier depends on the tier above it because it implements it.

CHEZ THIRUVATHUKAL

To Reinvent or not to Reinvent: Is It Even a Question?

Computer science is undergoing a bit of a transformation, with several departments across the country rethinking their CS curricula. A notable recent example is Georgia Tech, which introduced the notion of CS "threads." At Loyola University Chicago we, too, went through a similar exercise with the idea of broadening interest in CS—both at the undergraduate and graduate levels—and letting students explore more diverse trajectories. Not surprisingly, the US National Science Foundation (NSF) has recently issued a call for proposals to refresh the CS curriculum, realizing that there is indeed a need to rethink the way CS works stateside, seeing that interest isn't trending in the positive direction. Conversely, interest in CS, engineering, and IT is trending way up in China, India, and Latin/South America.

Redesigning undergraduate or graduate curricula won't change the landscape here in the US. I recall a visit with the Indian side of my family in December 1994, when I presented a paper on my dissertation research at what became known as the first high-performance computing conference in Bangalore, India. This visit left an indelible impression on me when I got a firsthand glimpse of what's happening there. My 12-year-old second cousin asked me, "Can you help me understand the running time of these three sorting algorithms?" Having developed an early interest in CS myself at age 14, which I considered late, I was impressed. But some of the questions she asked me were at the level I'd expect from advanced undergraduates or graduate students. What the heck was going on here? I pressed her to get an understanding of whether she was in a special program similar to what I experienced as a high-school student, in which you had to be good in mathematics just to sign up. She said something along the lines of, "No, these courses are available to everyone in my school. We all need to know this for our future, whether that future is here or elsewhere." I proceeded to help her with her homework questions and was delighted to see that Indian kids are learning this stuff and—more important—women are encouraged to do so as well! It's no secret that CS and engineering are experiencing ever-decreasing interest among the female population in the



package but physically place them in a separate source folder called `test`. The data-access tier tests, for example, are in the `points.hib` package in the `test` folder.

The Data-Access Tier

The `points.hib` package provides a Hibernate-specific implementation of the `points.dao.PointDAO` interface. As the dependency graph shows, this tier depends only on the interfaces in `points.dao` and `points.common`, which require no testing, thus we start here. Let's look first at the dependencies among the classes and interfaces in this package. The sinks in the intra-package dependency graph are the plain old Java objects (POJOs)—namely, `DefaultPoint`, which implements the `Point` interface, and `Color`, on which `DefaultPoint` depends. For convenience, we repeat the `Point` interface here:

```
public interface Point
extends Comparable<Point> {
  int getId();
  double getX();
  double getY();
  String getColor();
}
```

The `Color` class simply wraps around a color name implemented as a `String`:

```
class Color implements Comparable<Color> {
  private String name;
  public Color(String name) {
    this.name = name;
  }
  public String getName() { return name; }
  protected void setName(String name) {
    this.name = name;
  }
  // ...
}
```

Similarly, the `DefaultPoint` class is a straightforward implementation of the `Point` interface:

```
class DefaultPoint implements Point {
  private int id;
```

US, despite the enormous career potential. Like most academics in the field, I have no shortage of possible explanations; only time will tell whether any of them—mine included—are plausible. But I'm convinced there's something in my cousin's story that the NSF and the august institutions in America that teach CS might want to know and understand better: how do we get our secondary schools to realize that all our children should learn about computers and not just view them as glorified calculators?

I'm not convinced that CS—especially at the university level—needs to be reinvented; perhaps refined but not reinvented. I see little harm in CS departments exploring more creative programming and enhancing connections to science, business, and the arts. However, we must rethink education more generally and ask ourselves what ideas should be taught from an early age. I just recently started teaching a new course on the History of Computing as a core-knowledge course here at Loyola University Chicago. The history of computing and communication is a key component of "modern" world history, although you'd be hard pressed to find any historically focused course in secondary schools teaching these ideas and connecting them to the historical transformation that's literally happening before our eyes—or in the last century. Even a perusal of college history textbooks on western civilization produces nary a mention of these ideas. We as computing folk must do whatever we can to ensure the information age is considered as important as the industrial revolution. If the ideas of computing aren't worth teaching to anyone besides us, then why does it matter at all?

SI Units

In the last issue, we discussed unit testing and presented an example with symbolic expressions to represent units of measurement. The idea is to make it possible to carry the units of measurement the actual user is interested in maintaining. The code in this article was intended primarily for pedagogical purposes, but we're expanding it into a complete framework for dimensional analysis with full support for all known units of measurement. We'll see how far we can get and (I hope) report on it in a future column.

As part of this effort, we're also looking at an international standard called Système International d'Unites (SI), which breaks every quantity down into a combination of the dimensions mass, length (or position), time, charge, temperature, intensity, and angle. These SI units are a helpful tool for the problem at hand because they let you focus on the problem more abstractly and worry about the specific mapping to and from actual units later. Let's say I want to talk about an expression, meters/second^2. As "meters" fit into the SI notion of "length" or "position" and "second" into the notion of time, we can think of this expression's dimensions as length/time^2.

Note that it's still of paramount importance to address the problem of metric versus imperial units and to ensure that the units are consistent when operations are performed. The problem of adding "kg" and "pounds," for example, must be addressed when working with two different sets of measurement units. Even in the same system, we must take care to normalize the units to a common unit.

```
  private double x, y;
  // ...
}
```

We need a simple component-level unit test of the `Color` class. For brevity, we show only one test method, which tests the `Color.getName()` method:

```
public class TestColor extends TestCase {
  // ...
  public void testGetName() {
    Color c = new Color("orange");
    assertEquals("orange", c.getName());
  }
  // ...
}
```

We also need a simple test of the `DefaultPoint` class. Again, we show only one test method, but the complete test checks the various constructors, setter and getter methods, and comparison and other auxiliary methods:

```
public class TestDefaultPoint
extends TestCase {
  // ...
  public void testSetX() {
    DefaultPoint p = new DefaultPoint();
    p.setX(7);
    assertEquals(7, p.getX(), 0);
  }
  // ...
}
```

The preceding tests convince us that the POJOs work, so now it's time to look at the DAO itself. As you recall, the DAO interface looks like this, specifying the typical create, read, update, and delete (CRUD) functionality, as we would expect. Since the second hike,[2] we've converted our example code to Java Generics. Now, we can express more precisely that the result of the `findAll()` method is a collection of points rather than arbitrary objects:

```
public interface PointDAO {
  void init();
  Point create(double x, double y,
      String color);
  Point find(int id);
  Collection<? extends Point> findAll();
  void remove(int id);
  void update(int id, double x, double y,
      String color);
  Collection<String> findColors();
}
```

It's convenient to have a simple in-memory stub implementation of the DAO for two reasons: one, it provides a reference implementation for writing DAO unit tests, which we can test against until we're confident that the test itself is good enough before moving on to the real Hibernate-based implementation. Two, it lets us test some classes in the upper tiers without involving an actual database. The stub implementation simply stores the points in a hash map from the points' IDs (`Integer`) to the actual points. The implementation is straightforward, so we show only a few sample methods:

```
public class InMemoryDAO implements PointDAO {

  private Map<Integer, DefaultPoint> map =
    new HashMap<Integer, DefaultPoint>();

  private int currentId = 0;

  public synchronized Point create(double x,
      double y, String color) {
    ++ currentId;
    DefaultPoint p =
      new DefaultPoint(x, y,
        new Color(color));
    p.setId(currentId);
    map.put(currentId, p);
    return p;
  }

  public synchronized Point find(int id) {
    return map.get(id);
  }

  public synchronized void remove(int id) {
    map.remove(id);
  }

  // ...
}
```

The real Hibernate-based implementation is still the same as in the second hike.[2] It relies heavily on Spring in two ways: the `HibernateDaoSupport` superclass painlessly exposes Hibernate's functionality, and the `ApplicationContextAware` interface explicitly tells Spring that this class needs to look up other components:

```
public class HibernatePointDAO
extends HibernateDaoSupport
implements PointDAO, ApplicationContextAware
{
  // ...
  protected Point createPoint(double x,
      double y, Color color) {
    Point pt = new DefaultPoint(x, y, color);
    getHibernateTemplate().save(pt);
    return pt;
  }

  public Point create(double x, double y,
      String color) {
    Color col = (Color) getHibernateTemplate()
      .get(Color.class, color);
    return createPoint(x, y, col);
  }

  public Point find(final int id) {
    return (Point) getHibernateTemplate()
      .get(DefaultPoint.class,
        new Integer(id));
  }

  // ...
}
```

To build confidence in the correctness of these DAOs, we'll want to test them with increasingly complex scenarios. We start by unit testing the in-memory version of the DAO, making sure first that a freshly created DAO is empty and then checking whether we can find a point after creating it. Figure 3 shows an example of this.

We continue by testing the real, Hibernate-based DAO implementation. This integration test requires proper access to the persistent data source to which Hibernate maps our POJOs. In the Web application itself, we use Spring to manage the various dependencies: the DAO depends on a Hibernate session factory, which depends on a Java Database Connectivity (JDBC) data source. The challenge is to write the test case in a way that's consistent with our decision to use Spring—that is, we also want Spring to inject dependencies into the test.

Fortunately, this is a common situation, and Spring's `AbstractTransactionalDataSourceSpringContextTests` class (perhaps we should call it A47S) is exactly the right superclass for it. In the `getConfigLocations()` method, we tell Spring where to look for the application context definition files in which we specify the test case's various dependencies. The test case's application context definition file is almost the same as the one for the Web application, except the former must refer directly to a local data source, whereas the latter can use the servlet container to find the data source via the Java Naming and Directory Interface (JNDI). Concretely, this is just a minor change to the data source beans in the application context file.

The A47S class provides automatic transaction rollback, meaning we can test it against the production database without having to worry about modifying it. Because we're running in a test environment without real data, we explicitly reinitialize the DAO to the empty state before running each test method. We could also use DBUnit (http://dbunit.sourceforge.net/) to prepopulate the database to a known state before testing.

Unsurprisingly, the rest of the test case is the same as for the in-memory DAO stub:

```
public class TestHibernatePointDAO
extends AbstractTransactionalDataSource-
    SpringContextTests {

  private PointDAO dao;

  public void setPointDAO(PointDAO dao) {
    this.dao = dao;
  }

  protected String[] getConfigLocations() {
    return new String[] {
      "classpath:points/hib/" +
      "applicationContextHibernate.xml"
    };
  }

  protected void onSetUpInTransaction()
  throws Exception {
    dao.init();
  }

  // same test cases as TestInMemoryDAO
}
```

If we run the JUnit tests in Eclipse, we get the test results in the usual JUnit Eclipse view. Figure 4 shows the test results in the `points.hib` package.

The Business Tier

The `points.biz` package contains the interface and default implementation of the service object that implements specific business functionalities on top of the data-access tier. The service object's interface is

```
public interface RegressionService {
  void reset();
  void addPoint(double x, double y,
      String color);
  Point getPoint(int id);
  RegressionResult getResult();
  void removePoint(int id);
  void editPoint(int id, double x, double y,
      String color);
  Collection<String> getColors();
}
```

In this interface's implementation, the CRUD methods more or less directly invoke the corresponding DAO methods while the `getResult()` method performs the actual linear regression. The result is an instance of a transfer object interface:

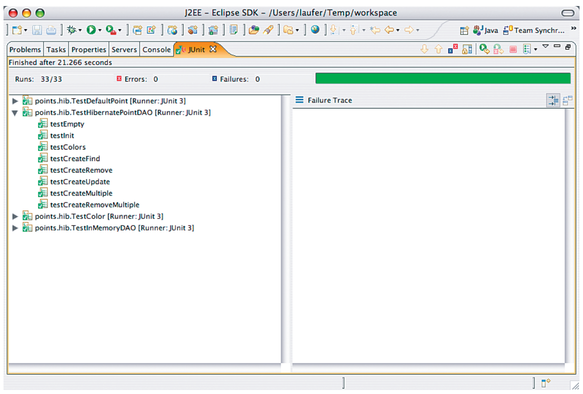
```
public interface RegressionResult {
  Collection<? extends Point> getPoints();
  double getSlope();
  double getYIntercept();
}
```

Using plain JUnit tests, we can test the service against the in-memory stub object defined as part of the DAO tests. As usual, we try to test the functionality thoroughly, working our way from simple to complex scenarios (see Figure 5).

Assuming you're a Spring fan by now, you'll have noticed that we explicitly set the DAO for the `DefaultRegressionService` to be tested, which is acceptable for our purposes. But if you're wondering whether Spring can inject this dependency for us, the answer is yes. Spring provides

```
public class TestInMemoryDAO
extends TestCase {

  private PointDAO dao;

  protected void setUp() throws Exception {
    super.setUp();
    dao = new InMemoryDAO();
  }

  protected void tearDown() throws Exception {
    dao = null;
    super.tearDown();
  }

  public void testEmpty() {
    assertTrue(dao.findAll().isEmpty());
  }

  public void testInit() {
    dao.init();
    assertTrue(dao.findAll().isEmpty());
  }

  public void testColors() {
    Collection<String> colors =
      dao.findColors();
    assertEquals(3, colors.size());
    assertTrue(
      colors.contains(Color.RED.getName()));
    // same for other predefined colors
  }

public void testCreateFind() {
  int size = dao.findAll().size();
  Point p =
    dao.create(2, 3, Color.RED.getName());
  assertEquals(size + 1, dao.findAll().size());
  assertEquals(p.getId(),
    dao.find(p.getId()).getId());
  assertTrue(dao.findAll().contains(p));
}

public void testCreateRemove() {
  int size = dao.findAll().size();
  Point p =
    dao.create(2, 3, Color.RED.getName());
  assertEquals(size + 1, dao.findAll().size());
  assertEquals(p.getId(),
    dao.find(p.getId()).getId());
  assertTrue(dao.findAll().contains(p));
  dao.remove(p.getId());
  assertEquals(size, dao.findAll().size());
  assertTrue(dao.find(p.getId()) == null);
  assertFalse(dao.findAll().contains(p));
}

public void testCreateUpdate() {
  int size = dao.findAll().size();
  Point p =
    dao.create(2, 3, Color.RED.getName());
  assertEquals(size + 1,
    dao.findAll().size());
  assertEquals(p.getId(),
    dao.find(p.getId()).getId());
```

Figure 3. Example of `TestInMemoryDAO`. This code unit-tests the in-memory stub implementation of the data access object (DAO) using increasingly complex scenarios.



```
Finished after 21.266 seconds
Runs:  33/33     Errors:  0     Failures:  0

▶ points.hib.TestDefaultPoint [Runner: JUnit 3]
▼ points.hib.TestHibernatePointDAO [Runner: JUnit 3]
    testEmpty
    testInit
    testColors
    testCreateFind
    testCreateRemove
    testCreateUpdate
    testCreateMultiple
    testCreateRemoveMultiple
▶ points.hib.TestColor [Runner: JUnit 3]
▶ points.hib.TestInMemoryDAO [Runner: JUnit 3]
```

Figure 4. Test results in the Eclipse JUnit view. This view shows the outcome of a particular test run as a tree.

another class with a fairly long name, `AbstractDependencyInjectionSpringContextTests` (A45S?), for test cases that obtain their dependencies from an application context; we can rewrite the previous test case by extending this class:

```
public class TestRegSvcInMemorySpring
extends AbstractDependencyInjection-
    SpringContextTests {
  private RegressionService service;

  public void setRegressionService(
      RegressionService service) {
    this.service = service;
  }
  protected String[] getConfigLocations() {
    return new String[] {
```

```
    assertTrue(dao.findAll().contains(p));
    dao.update(p.getId(), 4, 5,
      Color.GREEN.getName());
    Point q = dao.find(p.getId());
    assertEquals(p.getId(), q.getId());
  }

  public void testCreateMultiple() {
    Collection<Point> points =
      new HashSet<Point>(dao.findAll());
    int size = points.size();
    Point p1 =
      dao.create(2, 3, Color.RED.getName());
    Point p2 =
      dao.create(3, 4, Color.BLUE.getName());
    Point p3 =
      dao.create(4, 5, Color.GREEN.getName());
    assertTrue(p1.getId() != p2.getId());
    assertTrue(p2.getId() != p3.getId());
    assertTrue(p1.getId() != p3.getId());
    assertEquals(size + 3,
      dao.findAll().size());
    for (Point p :
        new Point[] { p1, p2, p3 }) {
      assertEquals(p.getId(),
        dao.find(p.getId()).getId());
      assertTrue(dao.findAll().contains(p));
      assertFalse(points.contains(p));
    }
  }

  public void testCreateRemoveMultiple() {
    Collection<Point> points =
      new HashSet<Point>(dao.findAll());
    int size = points.size();
    Point p1 =
      dao.create(2, 3, Color.RED.getName());
    Point p2 =
      dao.create(3, 4, Color.BLUE.getName());
    Point p3 =
      dao.create(4, 5, Color.GREEN.getName());
    assertTrue(p1.getId() != p2.getId());
    assertTrue(p2.getId() != p3.getId());
    assertTrue(p1.getId() != p3.getId());
    assertEquals(size + 3,
      dao.findAll().size());
    for (Point p :
        new Point[] { p1, p2, p3 }) {
      assertEquals(p.getId(),
        dao.find(p.getId()).getId());
      assertTrue(dao.findAll().contains(p));
      assertFalse(points.contains(p));
    }
    for (Point p :
        new Point[] { p1, p2, p3 }) {
      dao.remove(p.getId());
    }
    assertTrue(
      points.containsAll(dao.findAll()));
    assertTrue(
      dao.findAll().containsAll(points));
  }
}
```

```
      "classpath:points/hib/" +
        "applicationContextInMemory.xml",
      "classpath:points/biz/" +
        "applicationContext.xml"
    };
  }

  public void onSetUp() { service.reset(); }

  public void onTearDown() { service = null; }
  // test methods exactly as before
}
```

To specify this test case's dependencies, we use two application context files, one that specifies which DAO to use and one that is independent of DAO choice. Each is very simple:

```
<beans>
  <bean id="pointDao"
  class="points.hib.InMemoryDAO"/>
</beans>

<beans>
  <bean id="regressionService"
  class="points.biz.DefaultRegressionService">
    <property name="pointDAO" ref="pointDao"/>
  </bean>
</beans>
```

To integration-test the service object against the Hibernate-based DAO, we do only two things: extend `AbstractTransactionalDataSourceSpringContextTests` and use an application context definition file that provides the Hibernate-based DAO as shown in the second hike:[2]

```
public class TestRegSvcHibernate
extends AbstractTransactionalDataSource-
    SpringContextTests {
```

```
public class TestRegSvcInMemory
extends TestCase {

  private DefaultRegressionService service;

  protected void setUp() throws Exception {
    super.setUp();
    service = new DefaultRegressionService();
    service.setPointDAO(new InMemoryDAO());
  }

  protected void tearDown() throws Exception {
    service = null;
    super.tearDown();
  }

  public void testAddTwo() {
    service.addPoint(1, 3, "red");
    service.addPoint(2, 4, "blue");
    RegressionResult result =
      service.getResult();
    assertEquals(2, result.getPoints().size());
    assertEquals(2.0, result.getYIntercept());
    assertEquals(1.0, result.getSlope());
  }

  public void testAddRemoveMultiple() {
    service.addPoint(1, 2, "red");
    service.addPoint(3, 4, "red");
    service.addPoint(4, 5, "red");
    RegressionResult r1 = service.getResult();
    assertEquals(3, r1.getPoints().size());
    for (Point p :
        new ArrayList<Point>(r1.getPoints())) {
      service.removePoint(p.getId());
    }
    RegressionResult r2 = service.getResult();
    assertEquals(0, r2.getPoints().size());
  }

  public void testReset() {
    service.addPoint(1, 2, "red");
    service.addPoint(3, 4, "red");
    service.addPoint(4, 5, "red");
    RegressionResult r1 = service.getResult();
    assertEquals(3, r1.getPoints().size());
    service.reset();
    RegressionResult r2 = service.getResult();
    assertEquals(0, r2.getPoints().size());
  }

  // ...
}
```

Figure 5. `TestRegSvcInMemory` **example. This code unit-tests the default implementation of the registration service against the in-memory stub implementation of the DAO.**

```
  private RegressionService service;

  public void setRegressionService(
    RegressionService service) {
    this.service = service;
  }

  protected String[] getConfigLocations() {
    return new String[] {
      "classpath:points/hib/" +
        "applicationContextHibernate.xml",
      "classpath:points/biz/" +
        "applicationContext.xml"
    };
  }

  protected void onSetUpInTransaction()
      throws Exception {
    service.reset();
  }
  // test methods exactly as before
}
```

The Web Tier

The `points.web` package contains the Struts (http://struts.apache.org) actions that provide the application's dynamic behavior. In the Struts Web application framework, actions are small components that plug into the controller, which makes them hard to test in isolation using plain JUnit. `AddAction`, for example, adds a point to the current collection and expects to receive the incoming data as part of a Struts form, among other dependencies on the Struts controller servlet:

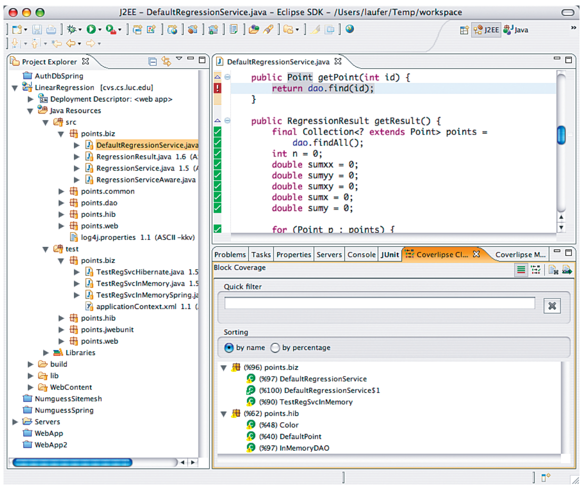
```
public class AddAction
extends RegressionServiceActionSupport
implements Constants {

  public ActionForward execute(...) ... {

    // obtain arguments from form bean
    double x = Double.parseDouble(BeanUtils
      .getProperty(pointForm, PROPERTY_X));
    double y = Double.parseDouble(BeanUtils
      .getProperty(pointForm, PROPERTY_Y));
    String color = BeanUtils.getProperty(
      pointForm, PROPERTY_COLOR);

    // interact with model
    getRegressionService()
      .addPoint(x, y, color);

    request.setAttribute(
      ATTRIBUTE_MESSAGE_KEY, "add.message");
    return mapping.findForward(
      FORWARD_SUCCESS);
  }
}
```

The StrutsTestCase framework (http://strutstestcase.sourceforge.net) is an extension of JUnit that supports the testing of Struts actions either in isolation or in their natural habitat (a servlet container). Here, we focus on testing outside of the container, which is generally preferable for performance and vendor independence. For this purpose, the StrutsTestCase framework provides the MockStrutsTestCase class, which simulates a Struts controller servlet running in a servlet container. Our test case extends this class; for brevity, we show only the test method that tests the AddAction. The framework's methods specifically support the testing of Struts actions.

Because the StrutsTestCase framework has nothing to do with the Spring framework, but Spring manages our Struts actions, we must tie things together rather carefully in the setUp() method. First, we need a Struts configuration file that refers to suitable Spring application context files. Second, we need to know where the Spring–Struts connection—the ContextLoaderPlugin—puts the Spring application context in the (simulated) servlet context. Finally, we pull the linear regression service object out of the Spring application context and are ready to go:

```
public class TestActionsInMemory
extends MockStrutsTestCase {

  RegressionService service;

  public void setUp() throws Exception {
    super.setUp();
   setContextDirectory(
     new File("WebContent"));
   setConfigFile("/WEB-INF/struts/test/" +
     "struts-config-inmemory.xml");

  getActionServlet().init();

  WebApplicationContext ctx =
    (WebApplicationContext) getSession()
    .getServletContext()
    .getAttribute(ContextLoaderPlugIn
      .SERVLET_CONTEXT_PREFIX);
  service = (RegressionService)
    ctx.getBean("regressionService");
  service.reset();
 }

 public void tearDown() throws Exception {
  service = null;
  super.tearDown();
 }

 public void testAddSubmit() {
  int size =
    service.getResult().getPoints().size();
  setRequestPathInfo("/addSubmit");
  addRequestParameter("x", "7.7");
  addRequestParameter("y", "8.8");
  addRequestParameter("color", "red");
  actionPerform();
  verifyForward("success");
  verifyNoActionErrors();
  Collection<? extends Point> points =
    service.getResult().getPoints();
  assertEquals(size + 1, points.size());
  Point p = null;
  for (Point q : points) { p = q; }
  assertEquals(7.7, p.getX());
  assertEquals(8.8, p.getY());
  assertEquals("red", p.getColor());
 }

 // similar methods to test other
 // actions and scenarios
}
```

The only difference between testing against the in-memory stub DAO and the Hibernate-based DAO is in the corresponding Spring application context file. It's convenient to have two Struts configuration files for this purpose, one that refers to the application context with the



```
    public Point getPoint(int id) {
        return dao.find(id);
    }

    public RegressionResult getResult() {
        final Collection<? extends Point> points =
            dao.findAll();
        int n = 0;
        double sumxx = 0;
        double sumyy = 0;
        double sumxy = 0;
        double sumx = 0;
        double sumy = 0;

        for (Point p : points) {
```

```
▼ (%96) points.biz
      (%97) DefaultRegressionService
      (%100) DefaultRegressionService$1
      (%90) TestRegSvcInMemory
▼ (%62) points.hib
      (%48) Color
      (%40) DefaultPoint
      (%97) InMemoryDAO
```

Figure 6. Coverlipse plug-in for Eclipse. The tool helps us visualize the code coverage in JUnit tests, both at the Java source level and as a percentage by package/class.

in-memory DAO and one to the application context with the Hibernate-based DAO.

Code Coverage: How Well Are We Doing?

In the last issue,[3] we mapped out the unit-testing design space. Among other things, we discussed our testing strategy's transparency, ranging from black box to gray box to white box. If we follow a pure black-box approach, we'll be satisfied by testing all the methods in the interface with various scenarios including boundary values. But we can't be sure that we tested each and every line of code of the classes at least once.

If we take more of a white-box perspective, however, we can use a code coverage tool to find out how thorough our tests really are—for example, Coverlipse (http://coverlipse.sourceforge.net) is an Eclipse plug-in that collects code coverage data seamlessly during JUnit testing. It measures all-uses coverage and statement coverage and visualizes the collected data in two ways: source files with lines covered or not covered, and percentages of coverage by package and class. For example, as the upper-right part of Figure 6 shows, we've forgotten to test the `getPoint()` method in the `DefaultRegressionService` class. Consequently, our statement coverage for this class is below 100 percent, with coverage of the next-lower tier even less complete.

Acceptance-Testing the Web Application

So far, we've focused our testing efforts on interacting directly with the components to be tested, either unit-testing them in isolation or integration-testing them with other components. This approach is highly valuable for increasing our confidence in component correctness, but we haven't yet tested the entire system as the user experiences it. In particular, we haven't tested the user interface's view tier. In this section, we'll look at two related approaches for automated acceptance testing of the entire Web application: programmatic testing and browser-based testing. Both interact with the application via HTTP.

Programmatic Acceptance Testing

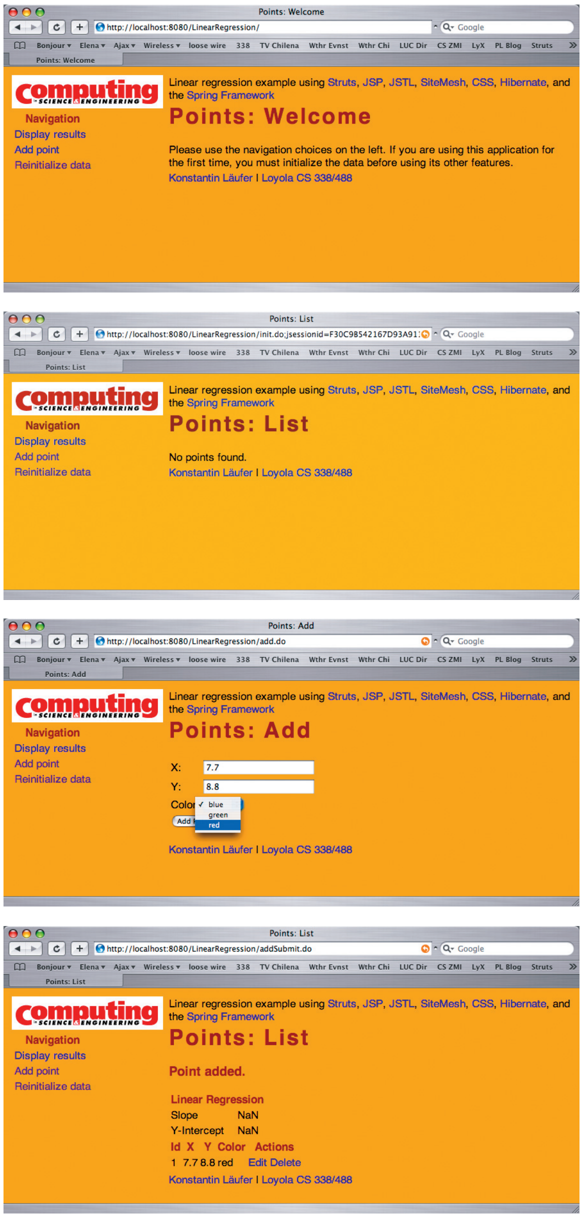
In programmatic acceptance testing, our test cases simulate interaction with the Web application by playing the role of a user agent (such as a Web browser). In a typical test case, we navigate to a page, follow a link (or fill out and submit a form) on the page, and check the response for correctness.

Several frameworks support this approach at different levels of abstraction. We'll look specifically at jWebUnit (http://jwebunit.sourceforge.net), which has several advantages. Above all, it supports relatively high-level interaction with the Web application to be tested, so test cases can stay concise and easy to understand.

Moreover, jWebUnit can use the same resource bundles as the Web application under test, which is tremendously useful. By following the approach outlined in the previous hike[2] and avoiding hard-coded text in the views in favor of property files (such as Java resource bundles), we can easily internationalize or otherwise change pieces of text in the views. By setting up our test cases to use the same resource bundles, the test cases become immune to such changes as long as they're done via resource bundles.

As an example, let's consider the typical scenario of adding a point to the regression, illustrated as a sequence of screenshots in Figure 7. To test this scenario using jWebUnit, we extend `WebTestCase`. In the `setUp()` method, we specify the base URL of the Web application under test as well as the application resources (property file) used to look up any text strings defined as properties. Closely following the scenario, we start with the application's default page, then click the link for adding a point, fill out and submit the resulting form, and verify that the application has added the point to the current collection. The jWebUnit methods support these steps rather directly:

```
public class TestAcceptance
extends WebTestCase {

  public void setUp() throws Exception {
    super.setUp();
   getTestContext().setBaseUrl(
     "http://localhost:8080/" +
        "LinearRegression");
    getTestContext().setResourceBundleName(
      "points/web/ApplicationResources");
  }

  public void tearDown() throws Exception {
    super.tearDown();
  }

  public void testAddPoint() {
    // start at the default page
    beginAt("/");
    assertKeyPresent("index.heading");
    // follow the "Add point" link
    clickLinkWithText(
      getMessage("navigation.link.add"));
    assertKeyPresent("add.heading");
    // fill out and submit the form
    setTextField("x", "7.7");
    setTextField("y", "8.8");
    selectOption("color", "red");
    submit();
    // ensure that the point was added
    assertKeyPresent("add.message");
    assertTablePresent("pointsTable");
    assertTextInTable("pointsTable",
      new String[] {
        "1", "7.7", "8.8", "red" });
  }

  // similar methods to test other
  // scenarios
}
```



Points: Welcome

http://localhost:8080/LinearRegression/

Linear regression example using Struts, JSP, JSTL, SiteMesh, CSS, Hibernate, and the Spring Framework

Points: Welcome

Navigation
Display results
Add point
Reinitialize data

Please use the navigation choices on the left. If you are using this application for the first time, you must initialize the data before using its other features.

Konstantin Läufer | Loyola CS 338/488

Points: List

http://localhost:8080/LinearRegression/init.do;jsessionid=F30C9B542167D93A91

Linear regression example using Struts, JSP, JSTL, SiteMesh, CSS, Hibernate, and the Spring Framework

Points: List

Navigation
Display results
Add point
Reinitialize data

No points found.

Konstantin Läufer | Loyola CS 338/488

Points: Add

http://localhost:8080/LinearRegression/add.do

Linear regression example using Struts, JSP, JSTL, SiteMesh, CSS, Hibernate, and the Spring Framework

Points: Add

Navigation
Display results
Add point
Reinitialize data

X: 7.7
Y: 8.8
Color ✓ blue / green / red
Add

Konstantin Läufer | Loyola CS 338/488

Points: List

http://localhost:8080/LinearRegression/addSubmit.do

Linear regression example using Struts, JSP, JSTL, SiteMesh, CSS, Hibernate, and the Spring Framework

Points: List

Navigation
Display results
Add point
Reinitialize data

Point added.

Linear Regression

Slope	NaN
Y-Intercept	NaN

Id	X	Y	Color	Actions
1	7.7	8.8	red	Edit Delete

Konstantin Läufer | Loyola CS 338/488

Figure 7. Screenshot sequence. Here, we add a point to the regression, which we're testing programmatically.

Browser-Based Acceptance Testing

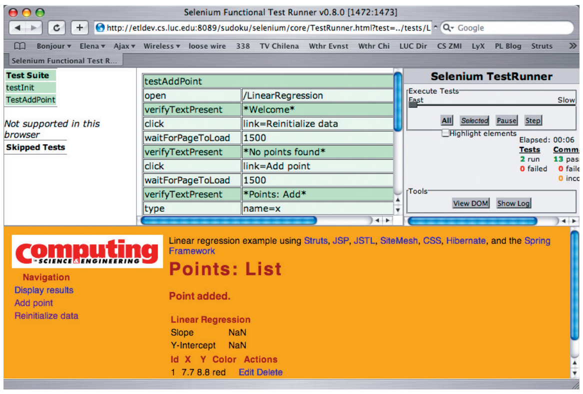
Seasoned Web developers will probably tell you that they spend a lot of time working out incompatibilities between browsers because of different JavaScript, document object model (DOM), or cascading style sheets (CSS) implementations. In some cases, the same browser behaves differently across platforms (most notably Internet Explorer). Consequently, the only way to iron out these differences is to test the Web application in the browser itself.

Enter Selenium (www.openqa.org/selenium), a testing tool for Web applications that runs test cases directly in the browser. This is the closest we can get to making an actual person execute the tests manually. Selenium is implemented in JavaScript, so the Web browser itself runs Selenium. The application runs in a separate iFrame and is manipulated via DOM/JavaScript, as Figure 8 shows.

Because the tests are run in JavaScript, Selenium must be hosted on the same server/port as the Web application, to comply with JavaScript's sandbox restrictions. Tests are created as HTML tables, which are discovered via the DOM and executed afterward. Test suites consist of tables as well, which refer to test cases (tables) via links.

Following our previous example, we create a suite referencing our two tests:

```
<table id="suiteTable" ... class="selenium">
```



Test Suite
testInit
TestAddPoint

Not supported in this browser
Skipped Tests

testAddPoint	
open	/LinearRegression
verifyTextPresent	*Welcome*
click	link=Reinitialize data
waitForPageToLoad	1500
verifyTextPresent	*No points found*
click	link=Add point
waitForPageToLoad	1500
verifyTextPresent	*Points: Add*
type	name=x

Selenium TestRunner
Elapsed: 00:06

Tests	Comm
2 run	13 pas
0 failed	0 faile
	0 inco

Linear regression example using Struts, JSP, JSTL, SiteMesh, CSS, Hibernate, and the Spring Framework

Navigation
Display results
Add point
Reinitialize data

Points: List

Point added.

Linear Regression
Slope NaN
Y-Intercept NaN

Id	X	Y	Color	Actions
1	7.7	8.8	red	Edit Delete

Figure 8. Selenium in action. The current test case is in the middle of the top half, and the test runner is controlled via the user interface in the top right section. Execution of each test case is animated, and the browser shows exactly what the test case is doing.

```
  <tbody>
    <tr><td><b>Test Suite</b></td></tr>
      <tr><td><a href="LinRegInit.html">
        testInit</a>
      </td></tr>
     <tr><td><a href="LinRegAddPoint.html">
        TestAddPoint</a>
      </td></tr>
  </tbody>
</table>
```

For the initialization test, we create the file `LinRegInit.html`:

```
<table ...>
  <tbody>
    <tr><td colspan="3">testInit<br></td></tr>
    <tr>
      <td>open</td>
      <td>/LinearRegression</td>
      <td>&nbsp;</td>
    </tr>
   <tr>
        <td>verifyTitle</td>
        <td>Points: Welcome</td>
        <td></td>
    </tr>
    <tr>
      <td>verifyBodyText</td>
      <td>*Welcome*</td>
      <td></td>
    </tr>
    <tr>
      <td>click</td>
      <td>link=Reinitialize data</td>
      <td></td>
    </tr>
    <tr>
      <td>waitForPageToLoad</td>
      <td>3000</td>
      <td></td>
    </tr>
    <tr>
        <td>verifyTitle</td>
        <td>Points: List</td>
        <td></td>
    </tr>
    <tr>
      <td>verifyBodyText</td>
      <td>*No*points*found*</td>
      <td></td>
   </tr>
  </tbody>
</table>
```

Note that after we click the link called "Reinitialize data," we instruct Selenium to wait for the page to load (with a timeout of three seconds). If this instruction isn't placed after each click/submit action, Selenium might report wrong results or downright fail because the iFrame might not have loaded, so Selenium is already trying to find/trigger elements inside it.

For our second test (`LinRegAddPoint.html`), we use HTML code that incorporates the table in Figure 9. It tests the same scenario as the jWebUnit acceptance test shown in the preceding subsection.

Once we finish writing the tests, we can run them in any browser that supports JavaScript. Selenium also offers an `unless` condition for use in the `<TR>` element to filter test suites known not to run in a particular browser:

```
<tr unless="browserVersion.isKonqueror ||
 browserVersion.isSafari">...</tr>
```

Additionally, Selenium now includes support for programmatic tests. In this mode, Selenium runs as a jWebUnit plug-in, so it's no longer necessary to write tests in HTML table form. The requirement, however, is that a Selenium server must run on a machine with local installations of the target browsers. Commands are sent via HTTP `GET` requests, which lets developers write base test classes for many languages, including Java. Deriving from these classes, we can write tests cases in the same way as all our other examples in this article.

testAddPoint		
open	/LinearRegression	
click	link=Reinitialize data	
waitForPageToLoad	1500	
verifyTextPresent	*No points found*	
click	link=Add point	
waitForPageToLoad	1500	
verifyTextPresent	*Points: Add*	
type	name=x	7.7
type	name=y	8.8
select	name=color	red
submit	id=pointForm	
waitForPageToLoad	1500	
verifyTextPresent	*Point added*	
verifyElementPresent	id=pointsTable	
verifyText	xpath=//table[@id='pointsTable']//tr[2]/td[1]	1
verifyText	xpath=//table[@id='pointsTable']//tr[2]/td[2]	7.7
verifyText	xpath=//table[@id='pointsTable']//tr[2]/td[3]	8.8
verifyText	xpath=//table[@id='pointsTable']//tr[2]/td[4]	red

Figure 9. Selenium test case for adding a point. The test case instructs the browser to navigate to the Web application, add a point, and verify whether the point has been added.

In this hike, we saw how to incorporate automated component, integration, and acceptance testing into the various tiers of a lightweight J2EE Web application architecture. However, numerous other frameworks and tools are available to help with testing Web-based and other applications in Java, including www.junit.org/news/extension/web; http://java-source.net/open-source/web-testing-tools; and http://java-source.net/open-source/code-coverage.

Although these architectures are widely used, there's a growing trend toward more compositional, loosely coupled architectures—for example, right now, service-oriented architectures (SOAs) are all the rage. In an upcoming hike, we plan to illustrate the SOA approach by starting with our original architecture, breaking down the business functionality into cohesive units, exposing them as services, and building more complex services and applications on top of them.

References

1. K. Läufer, "A Hike through Post-EJB J2EE Web Application Architecture," *Computing in Science & Eng.*, vol. 7, no. 5, 2005, pp. 80–88.
2. K. Läufer, "A Hike through Post-EJB J2EE Web Application Architecture, Part II," *Computing in Science & Eng.*, vol. 8, no. 2, 2006, pp. 86–94.
3. G.K. Thiruvathukal, K. Läufer, and B. Gonzalez, "Unit Testing Considered Useful," *Computing in Science & Eng.*, vol. 8, no. 5, 2006, pp. 76–87.

Konstantin Läufer is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, concurrent and distributed systems, mobile and embedded computing, human–computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him via http://people.cs.luc.edu/laufer.

George K. Thiruvathukal is an associate professor of computer science at Loyola University Chicago. His research interests include programming languages, operating systems, distributed systems, architecture and design, computing history, and enhancing science and computing education with emerging technologies. Thiruvathukal has a PhD from the Illinois Institute of Technology. He is a member of the ACM and the IEEE Computer Society. Contact him at gkt@cs.luc.edu or http://people.cs.luc.edu/gkt.

Benjamín González is a software developer intern at Hostway Corporation. He also works as a research assistant at Loyola University Chicago. His research interests include distributed systems, artificial intelligence, software architecture, operating systems, and Web development. González has an MS in computer science from Loyola University Chicago. Contact him at bgonzalez@cs.luc.edu.