Mission Made Possible
How one team harnessed tools and procedures to test a complex, distributed system during development

By Rex Black and Greg Kubaczkowski

Your mission should you choose to accept it: A client needs to implement an integrated, automated unit, component, and integration testing process. The system development teams are spread across 3 continents. They use multiple platforms. Their system architecture is interdependent and complex. Some tests need to be run on an application server. Both static and dynamic testing is required. The process you design must be simple to use, not an additional burden for busy developers. It should be automated and easily integrated. By the way, only 3 designers have been assigned to this mission. You have 6 weeks. Good luck. This message will self-destruct in 20 seconds…

Ever feel like you've been thrown into an episode of Mission Impossible? Even with test development tools at our disposal, designing and executing early test processes is never easy. Throw in short timelines and complex systems and it can seem truly impossible. But, this vital mission must be performed. Though this is only one unique situation, the problem solving tools and lessons learned have universal applications.

Mission Challenges

Challenge 1: Diverse Group; Multiple Platforms

The geographical distribution of the project arose in part from the fact that the company was growing through mergers and acquisitions. Testing, especially integration testing, was critical because the diverse and still-coalescing team would be learning each other's working and programming styles during the project. This created opportunities for miscommunication—and bugs. The team also had to contend with three different platforms: Linux, Solaris, and Windows. The tools we chose had to be flexible across these multiple platforms.

Challenge 2: Complex Architecture

Architecturally, the system under construction was composed of six layers (see figure 1). Each layer added additional services and capabilities used by the layers above it. In turn, each layer used services and capabilities provided by any layers below it. The increasing functionality and interdependence culminated at the GUI layer. The layers were built from components that provided one distinct service or capability, often by using services and capabilities provided by components at their own layer or below. The components, in turn, were built from units. A unit was the smallest distinct object that could be checked into the source code repository, written by a single programmer.

This system design, while elegant, not only constrained the definition of the testing tasks, but also the objectives and sequencing of those tasks. The technical challenge, then, was

for the test tool to be able to support the following three distinct categories of testing tasks (shown clockwise from the upper right in Figure 1):

- **U**nit testing: The testing of each unit, performed by interacting directly with its public and private interfaces. This testing would reveal logic and data flow errors in the basic building-blocks of the system themselves.

- **C**omponent testing: The testing of each component, performed by interacting with the services and behaviors of the component under test, which might in turn interact with other components. This testing would reveal problems in the services and behaviors of the constituent system components.

- **I**ntegration testing: The testing of two or more components, performed by interacting with the interfaces and behaviors of these collected components. The collected components could all reside in the same layer or in different layers. This testing would reveal incompatibilities and bottlenecks between related and dependent components.

Because of the three testing tasks the tool was to accomplish, we referred to it as the "UCI Test Harness."

Challenge 3: Interdependency of Test Sequencing

The constraints of the system design also influenced test task sequencing. The UCI Test Harness could test units independently of any other unit or component. However, before a component could be tested, three criteria had to be satisfied.
1. The programmers had to complete all the units in that component.
2. The programmers had to complete all the components in the layer beneath the one the component under test belonged to.
3. Within one layer, Programmers had to complete any component upon which the component under test relied.
The challenge was to define ways in which each of these testing activities could happen without interfering with each other and without adding any delays beyond those created by the clumsy test sequencing constraints imposed by the system architecture. One solution would have been to build potentially elaborate drivers, stubs, and other scaffolding to provide mock interfaces for testing. This would have been far too expensive and time-consuming to create; developers would have spent a large portion of their time writing test code that would later be thrown away. We chose instead to perform integration testing layer-by-layer, as the components that made up each layer were completed; and to wait until each layer was through with integration testing before performing integration testing on the next layer. (However, integration testing could include interfaces between components that spanned layers, as mentioned earlier.)
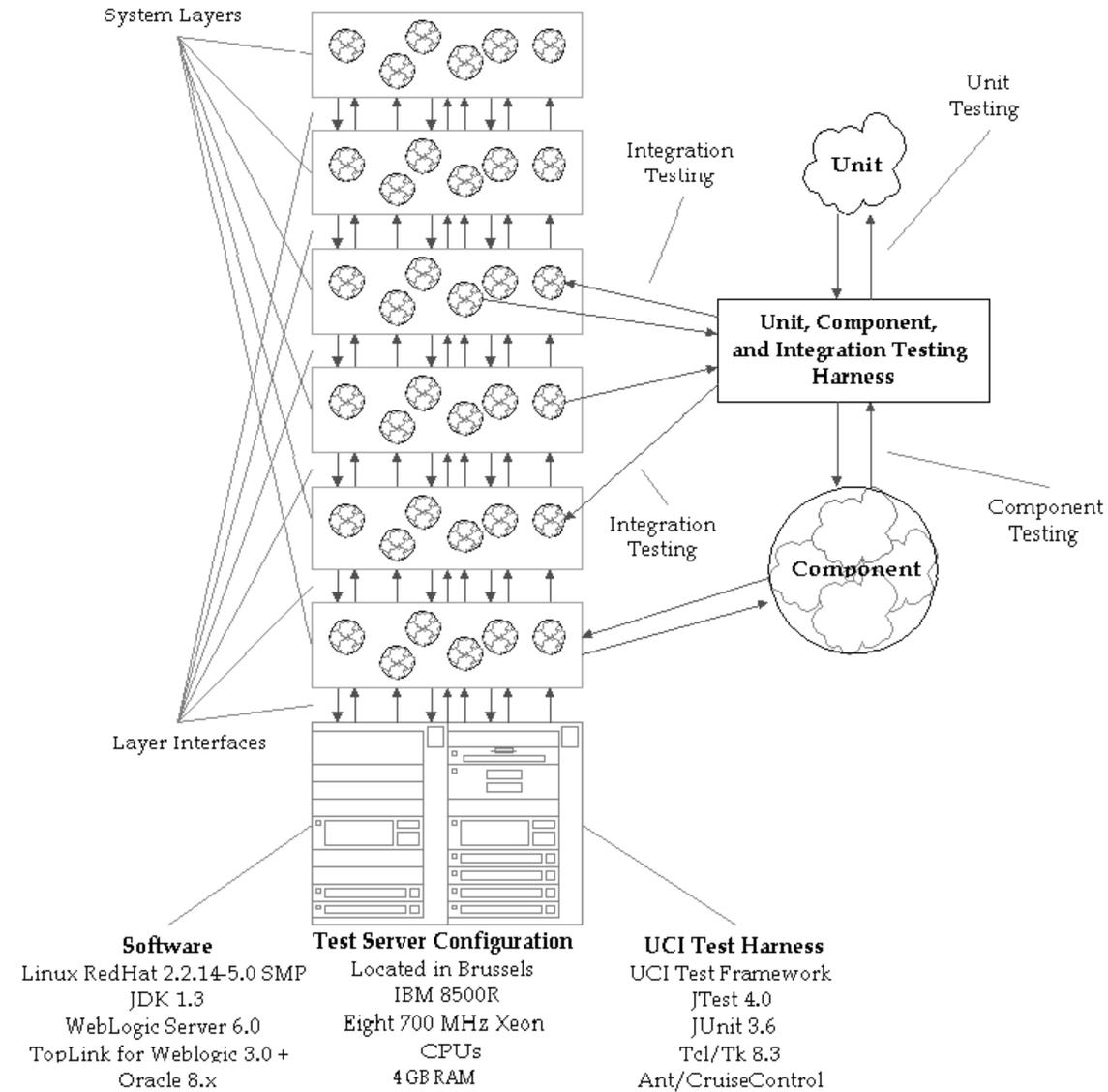
Figure 1: From Bottom Left, the Test Environment, the Architecture of the System Under Test, and the Unit, Component, and Integration Testing Activities Automated by the UCI Test Harness

Challenge IV: System Implementation Requiring an Application Server

Another challenge arose from the system implementation. The system was constructed in Java and Enterprise Java Beans (EJBs). EJBs, in order to provide the desired functionality, need to run within an application server (see the bottom of Figure 1). Although it would be possible to stub out the functionality provided by the server, that task would be almost as complicated as developing an application server itself. So, testing could only happen after the bean was deployed on the server. Once this was done, we could access the EJB through its remote interface and exercise it to test the desired execution sequences. The nature of the EJB architecture dictated the choices we made in our test tool selection process (see Test Tools sidebar).

Challenge V: Static and Dynamic Test Needs

We needed testing tools for the Java code that could perform both static and dynamic testing. Static testing consisted of looking for nonconformance with Java coding standards, as well as EJB specific coding standards.   Dynamic testing involved executing sequenced test steps with some predefined data and expected results. We had to provide facilities for this static and dynamic testing at all levels: unit, component and integration.

Challenge VI: Keep it Simple

The team needed processes that were not just effective and efficient but also uncomplicated and comfortable. For a geographically distributed team that was just gelling and still arranging itself at the management level, burdensome bureaucracy imposed from on high would never take root. So we needed a simple tool. It had to be simple for

- Programmers to write and run tests;
- The release engineering team to couple with automated builds;
- The project management team to check completeness of testing compliance and test results;
- The development staff and the testers responsible for the component/integration testing to use; and
- Uncomplicated automated execution and integration into the build process.

Challenge VII: Limited Resources and Time

There were only 3 designers tasked with developing testing tools and processes within the context of the overall effort. From the day we started on the project, we had six weeks to implement a working test tool and a workable test process for unit, component, and integration testing. We designated one person as toolsmith. The toolsmith would select the right tools, then prototype and build a test harness. The other two team members would focus on defining the process, training the development team in testing techniques, and building support for the effort.

Sidebar: Choosing the Tools

For the dynamic testing we chose Junit (www.junit.org), an Open Source framework for unit testing of Java code. It is 100% java based, which makes it natural for developers to use. They can easily create tests as Java classes, run them on their workstation, and group them into suites. The same tests can also be run on a separate test server.

For the static analysis of the Java code we chose Jtest. Jtest (www.parasoft.com) is a comprehensive, commercial java-testing tool. Although it can be used for dynamic testing also, it did not provide any EJB specific functionality, and thus did not offer any additional functionality beyond what JUnit already provided.

The build tool used for this project was Ant (jakarta.apache.org/ant/index.html). Ant is an open-source build tool written in Java that uses XML files to define the build targets.

Each target is composed of a set of tasks, which are implemented as java classes. Ant comes with a rich set of built-in tasks, including a junit task that allows incorporating the unit tests in the process, as well as ejbjar task for EJB deployment and packaging.

Mission Payoff

So, with all these challenge, why bother testing at this level? Why not just cancel the mission and go home? Three big payoffs made our mission critical.

- Budget: Bugs fixed in early testing are cheaper than bugs fixed in system testing. Would the geographically distributed teams develop components that would work together? Would changes to one component cause another component or integrated build to break? We didn't know. The chance of bugs was high. We calculated, conservatively, that testing early and often could help them find and fix over 1,000 bugs before they escaped into system testing—or worse, into shipping products—saving them hundreds of thousands, if not millions, of dollars. (For more on the financial consequences of bugs, see the references in StickyNotes.)

- Schedule: Late-breaking problems found just days before planned delivery dates can delay projects by weeks—or even result in outright project cancellation. In this case, a key round of investor funding was dependent on delivering the system on schedule, making delay entirely unacceptable. Thorough developer testing was a key part of reducing risk in this area.

- Quality. When programmer testing is overlooked in the hurly-burly of a rushed development effort, the crush of bugs found in system testing often results in just-barely-shippable systems. Eager—and demanding—customers awaited our client's next release. Good unit, component, and integration testing practices would lay the foundation for a quality product on the delivery date.

Saving money, hitting the schedule, and delivering quality products: This mission was not only possible. It was essential.

Mission Execution
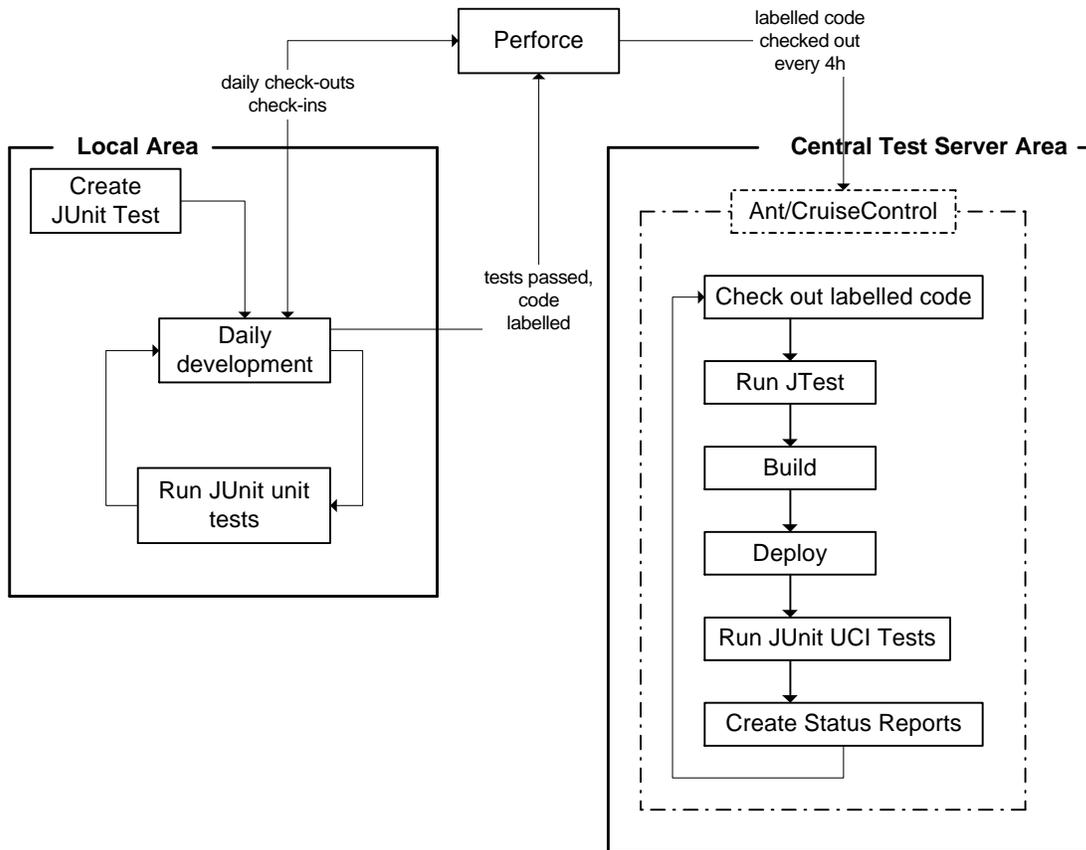
The Technical Elements

Figure 2: The Testing Process

To complete our mission, we developed a two-phase testing process. These two phases took place in two separate environments: the local development environment and the central test environment (see Figure 3). For the test server we chose Weblogic 5.1 as the application server, J2EE 1.1 specification for the java beans, Perforce as the source control system, Oracle 8 database, TopLink as the mapping tool for the persistency layer, and Ant as the build tool.

Phase I: Local Area Unit Tests

In the first phase, the developers of the java beans were responsible for the creation and execution of the unit tests against their own code on their workstations. Each workstation had a version of the application server, so the beans could be deployed and the tests executed. Once the developer was satisfied with the test results, the code in the repository would be marked with a floating label signifying that the code is believed bug free. Only these files would be used for the build on the test server.

Phase II: Central Test server UCI Tests

The second phase was the centralized unit/component/integration testing. This took place in an environment composed of a Linux based test server with Weblogic 5.1 and a separate database server, independent of the production database (see the bottom of Figure 1 again). The test process was driven by Ant coupled with CruiseControl (cruisecontrol.sourceforge.net) to provide a continuous build process. Every four hours, Ant checked the source control system (Perforce) for changes in the source files. If a programmer had made changes in any source file, Ant would detect these changes and trigger an automated build process.

The build process started by retrieving the source files, but only those marked with the floating label. (The rest were checked into the repository but considered unready for component or integration testing until they passed local unit testing.). The names of all source files that changed since the last build were logged, together with the name of the developer that did the change. This process was designed to help trace possible causes of test case failures. Next, Ant executed the Jtest static analysis. Once completed, the Java code was compiled. If the compilation went okay, the EJB was packaged and deployed. Ant provided the tasks for stopping and restarting the Weblogic application server to make this possible. This all occurred without any need for human intervention by the release engineering team.

After deploying the EJBs, Ant started dynamic testing of the beans via a JUnit task. The test cases were grouped by unit, component, and integration tests. Ant's JUnit task can be configured to run tests based on a certain naming convention. The naming convention allowed us to exclude some test cases from execution until certain conditions were met (e.g., intercomponent dependencies).

Once the tests were completed, the version of the source files used for this build was marked with a label based on a time stamp. A web-based report was created and posted on the web. This report listed the files that were added or changed since the last build, the test cases that were run, and the test results. An email notification was sent out to the interested parties. This functionality was provided by CuiseControl in this case, but can be easily implemented in Ant. This provided the compliance monitoring features for management, as well as making it easy for programmers to keep track of the status of their tests.

The Logistical Elements

Provide scope and strategies

One key element of defining the process was to outline the scope and strategies for unit, component, and integration testing. As mentioned earlier, the testing strategy included both static and dynamic elements. The static tests were based on predefined coding standards and rules for good Java code. The dynamic tests were designed using a combination of structural ("white box") and behavioral ("black box") techniques. For example, for structural tests, developers writing unit tests were to consider issues like code coverage, while those writing integration tests were to look at ways data and control

could flow across communicating components. For behavioral tests, developers were to design unit, component, and integration tests based on

1. Positive test cases where the item under test was to produce a result and
2. Negative test cases where the item under test was to detect an invalid condition and handle that condition gracefully.

Testing of basic performance evaluations (e.g., how long should it take to complete a particular service?) and internationalization was also within the scope. These were broad guidelines, within which each programmer, with the advice and consent of the appropriate technical lead, was free to make detailed decisions on the criteria that determined sufficiency of test coverage for the item being tested.

Outline developer role

For most programmers, this level of testing activity was a change to the routine programming process. For the company, it was a change to the way the programming organizations worked. Process changes are hard. We developed a plan to guide people through this process change that clearly spelled out roles, responsibilities, and timing. The plan included sections that assigned the activities discussed in both the plan and the design specification to particular participants in the unit, component, and integration test effort. See Figure 3 for an example. Note that this is not a vague statement of principles: "It is necessary to thoroughly test each piece of software," or "Testers should strive to find all bugs and help developers resolve them," or "Complete coverage of all functional requirements is desirable." Such ambiguous, non-actionable phrases, without any specific follow-up statements about how these goals are to be accomplished, sometimes occur in test plans destined for failure. Activities need to be clearly delineated, ownership assigned, and results measured and related back to a predetermined goal.

2.3.2 Component Test

Tools: JUnit

Activity	Actors
Define test criteria using Component Interface Specifications (CIS) and performance baseline data	Programmers, Project Leaders
Create fixtures	Programmers, Test Tool Engineer
Test in personal environment	Programmers
Add code & tests to repository (automatic execution)	Programmers, Test Tool Engineer
Analyze results	Programmers, Test Tool Engineer, UCI Test Manager, Project Leaders (escalation only)
Ongoing test maintenance	Programmers, Test Tool Engineer

The goal is to test 100% of CIS requirements. The UCI Test Manager will maintain traceability between functional requirements, tests, and prioritized risks to system quality based upon the risk analysis (Failure Mode and Effect Analysis) performed on the requirements.

Figure 3: A test plan excerpt showing roles and responsibilities

Provide Configuration and Support Plan

Part of the process also included configuring and supporting a test environment. Every programmer needed to have a personal test environment, which allowed the individual to write and run unit tests. The test plan also identified the people who were called upon to support this test environment.

Provide Escalation Plan

Even the best-defined processes, roles, and responsibilities can break down. It was important to define the escalation process. If any person was unable or unwilling to perform an assigned role or carry out assigned activities, this process outlined a clear path and process for the affected parties to take that issue up the management chain.

Provide Training

Having addressed the technical and logistical areas of the process, the time had come to introduce the programmers to writing test cases. We use the word "introduce" deliberately, because, while a few of the programmers had some experience with unit

testing, most did not. In addition, some of those who did have experience with testing described their testing strategy as, "Test *everything* that could break."

Since "everything" seemed as unachievable as "Prove that the software works," our presentation started with the fundamentals. We introduced the concept of risk-based testing. Risk-based testing allows the developers—and testers—a way to focus not on the infinity of things that could break, but rather on those areas that were either likely to break, dangerous when they did break, or critical to the customer that they not break. At that point, we moved on to techniques for testing, including structural testing tactics and integration strategies. We finished our basic presentation by talking about bug reporting and bug management.

Next, we moved on to specifics related to the processes we were proposing. This took the form of walking through the formal UCI test plan document we had created. The focus of this discussion was to make clear roles and responsibilities, along with the attendant benefits of these tasks. We needed to hand-off the project to the client. This meant that those we were handing the test plan to had to accept ownership.

Finally, we walked the attendees through the eleven-page UCI Test System Design Specification. This document included a description of the tools comprising the test system, sample test cases, and code samples. Such a document that can serve as a user's guide as well as an explanation of the test harness is invaluable when trying to get people to adopt a new process like this, but it must be simple to follow and simple to explain. At the same time, we gave them a hands-on demonstration of the entire test tool. As with the previous sessions, this session included many opportunities for questions and answers and audience interactivity.

This presentation was given twice, once in France and once in Belgium. The teams in California participated in the Belgium presentation by conference call. This allowed the presentation to reach all six teams involved.

Mission Results

No single process change or tool shows results overnight. Introducing tools, changing minds, and improving processes takes time. Two keys to process change success are to pick changes that will clearly contribute to project success and to make change achievable through small steps towards the goal. A major reason that the tools and processes turned out to be acceptable to the individual contributors is that though the changes we made were far-reaching and a departure from business as usual, they were lightweight in terms of cost and burden on the programmers. The change was major, but the steps to make the change were relatively easy to take. Baby steps are easier than great leaps. For example, no one had to attend two weeks of training on how to test; we just gave them the basics. No one had to do anything they weren't already supposed to do; they just had to coordinate those tasks in a particular way. No one had to change the way they worked profoundly; they just had to include tests of a particular type in the "test-first" approach they were using.

The technical tool we developed provided test results not just every now and then, but once, twice, or more every day. The very action that triggered risks—checking new code into the repository—would also trigger testing activities that would tell the management team whether the new code had created a problem. If problems were found, then the path for resolution was also clear: Fix the bugs and check the change back into the repository (triggering another test process). This loop would continue until the issue was resolved. Therefore, the process provided continuous feedback and opportunity for course correction on any problems with component quality, system integration, or regression, rather than waiting until the end. With the help of the testing harness and processes, our client delivered their system with minimal delays from the original schedule, realizing all of the critical benefits: budget, schedule, and quality. Mission accomplished.

Authors' Bios

Rex Black is the big dog at RBCS (www.rexblackconsulting.com). RBCS executes, trains, and advises on testing, test automation, and quality assurance projects for clients around the world.

Greg Kubaczkowski is a Test Engineer specializing in test process automation. He has worked on numerous projects covering all aspects of testing e-business and telecommunication applications.

Additional Notes

- The financial benefits of early testing are delineated in Black's *Managing the Testing Process, Second Edition* and Campenella's *Principles of Quality Costs*.
- Ant. Ant is an open-source build tool, written entirely in java. It is becoming the build tool of choice for java-based projects. For more information on Ant see:
  - Ant home page: http://jakarta.apache.org/ant/index.html
  - JavaWorld article "Automate your build process with Ant": http://www.javaworld.com/javaworld/jw-10-2000/jw-1020-ant.html
- JUnit. JUnit is an open-source java unit testing framework. All test cases are written in java, and can be easily combined into test suites and executed as part of regression testing process.
  - JUnit home page: http://www.junit.org/
  - Article from SUN on JUnit and EJBs: http://developer.java.sun.com/developer/technicalArticles/J2EE/testinfect/
- Jtest. Jtest is a commercial java testing tool. Some of its features include automatic black box testing, white-box testing, regression testing and static code analysis to ensure adherence to over 240 industry-respected coding standards.
  - Jtest home page: http://www.parasoft.com/jtest
- CruiseControl. CruiseControl is a tool for implementing a continuous build process.
  - CruiseControl home page: http://cruisecontrol.sourceforge.net/
- "Test Design: Developing Test Cases from Use Cases" (*STQE Magazine*, July/Aug 1999, Vol 1, Issue 4, p. 30–37):

Postscript: The Cactus Framework

With the growing number of server-side technologies, the testing process and tool selection becomes increasingly more difficult. J2EE is no exception. Projects based on this technology would include ejbs for the business logic and persistency layers, and a combination of java GUI clients and servlets and JSPs for the presentation layers.

The simplest way to test your ejbs is to use JUnit framework and access the beans from the client perspective. This approach, however, has a couple of serious drawbacks. It is often the case that the ejbs are called by server-side components, such as JSPs and servlets. Therefore, tests run from the client perspective are executed in a different environment then the production settings and can lead to different results. Additionally, if the project uses local interfaces (EJB2.0), then the ejbs can only be called by other server-side components that are running in the same jvm. Moreover, if the project uses servlets and JSPs components, an additional tool is needed for testing. This, of course, increased the complexity and maintainability of the test process.

To answers the problems described above, a new testing framework was developed. This framework is called Cactus and is an open-source initiative. Cactus is build upon JUnit, and uses servlets to execute testcases on the server side. This approach eliminates the limitation of the client side ejb testing approach and therefore more realistically reflects the production environment. Cactus also allows for testing other server-side technologies such as servlets, JSPs, Tag Libs and Filters.

The Cactus framework can be used either from an IDE or as part of the build process when combined with Ant build tool. It is a single tool solution to server-side testing needs. For more information on Cactus, see: http://jakarta.apache.org/cactus/index.html.