Published on **ONJava.com** (http://www.onjava.com/)
http://www.onjava.com/pub/a/onjava/2004/01/21/jdepend.html
See this if you're having trouble printing code examples

Managing Your Dependencies with JDepend

by Glen Wilcox
01/21/2004

As a developer and architect, I'm always on the lookout for tools that will quickly provide feedback on the quality of software architectures and designs. The problem is that most measures of architectural and design quality tend to be vague qualities — scalability, reliability, maintainability, flexibility, modularity, etc. — that are difficult to measure in a repeatable, quantitative sense.

In this article, I'll introduce you to JDepend, a freely available tool that can provide insight into several qualities of your software architecture. JDepend analyzes the relationships between Java packages using the class files. Since packages represent cohesive building blocks of your architecture, maintaining a well-defined package structure provides insight into architectural qualities of maintainability, flexibility, and modularity. Packages also provide a useful mechanism for estimating the impact of requirements changes, so understanding their dependencies is also useful in this respect as well. Because JDepend's metrics are based on class files, they can be used to track the true state of your architecture at any given point in the software lifecycle.

Finally, I'll use JDepend to analyze Sun's J2EE Java Pet Store to give you an idea of how to utilize JDepend to manage your software development efforts.

A Little OO Background

Before we dive into the specifics of JDepend, I though it would be a good idea to review a few OO design concepts to set the stage for the rest of our discussion. A logical question is: "*Why not measure dependencies between classes instead of dependencies between packages*"? The problem with using classes is that OO design is predicated upon the notion of autonomous (or semi-autonomous) groupings of state and behavior (classes) collaborating to perform work. So at some level, you want to have classes working together or depending on each other. Unfortunately, the number of classes tends to get large quickly. If you pick an arbitrary maximum size for a class, say 300 lines of code (LOC), building a system of 300,000 LOC will give you a minimum of 1000 classes. Very quickly, you see that managing dependencies at the class level will not work. Packages provide a reasonable alternative. Classes that collaborate closely to perform a specific task are referred to as "cohesive" and they get grouped together into a package. The same technique can be applied to packages that collaborate closely; they form "subsystem" packages. You can now focus on managing the dependencies between the packages, which turns out to be a more realistic goal.

So how do we go about managing dependencies between packages? As a starting point, let us look at the differences between the two primary types of classes: concrete and abstract. In Java, a concrete class is any class that can be directly created using the new operator. The type of these classes is fixed when the code is compiled and at some point in the system, every object is represented by a concrete class. A dependency occurs when one class uses another concrete class within its implementation. This basically makes the statement "*my implementation depends on this concrete type*." As experience in developing OO systems has evolved, designs that minimize the dependencies on concrete types have proven to be the most flexible. This flexibility is achieved through the use of abstract classes (this includes abstract classes, as well as interfaces in Java). By utilizing abstract classes in your implementation, you allow your class to accept any class that implements the contract defined by the abstract class. You see this pattern at work in the J2SE and J2EE APIs, as well as other design patterns. So managing package dependencies boils down to minimizing your use of concrete classes defined in other packages.

Getting and Using JDepend

JDepend is an open source software program available for download at www.clarkware.com. The download includes the source code, JUnit test cases, documentation, pre-build .jar files, an ANT build script, and a sample application for testing purposes. The online documentation

covers all of the features available for setting up, configuring, and running JDepend so I'll just highlight a few of the key features here.

After unzipping the download file, you can verify your download by running JDepend on the sample application provided in the download. Assuming you are in the directory where you unzipped the files, execute the following command:

```
java -cp ./lib/jdepend.jar jdepend.swingui.JDepend ./sample
```

This should produce a window similar to the one shown in Figure 1.

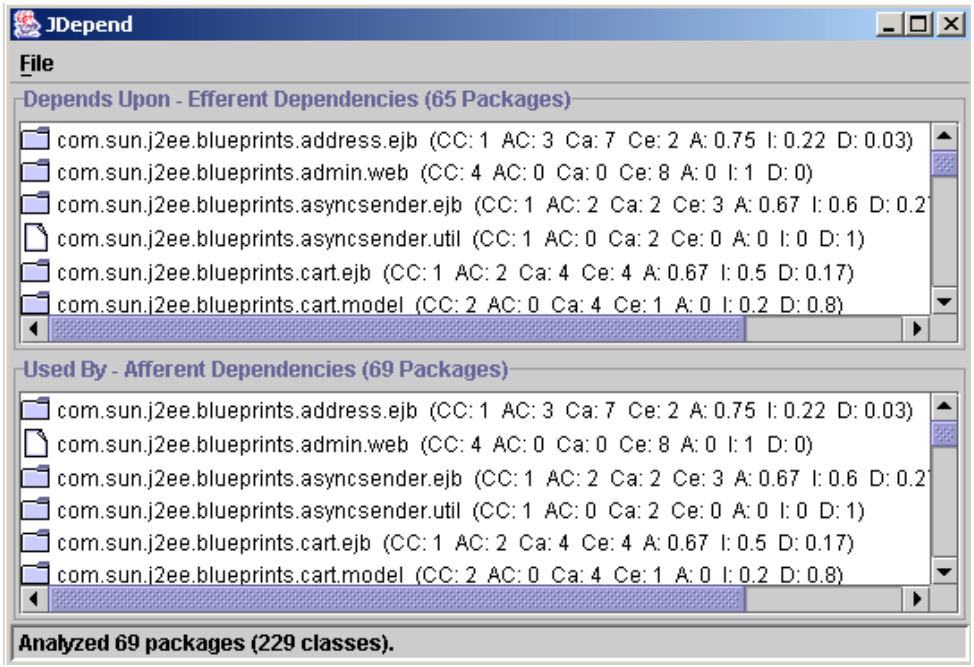

Figure 1. JDepend's Swing GUI

This is JDepend's Swing GUI. For a target package, the GUI provides drill-down capability for the packages it depends upon and the packages that depend upon it. The GUI also provides a series of metrics for each package. We will talk more about the metrics reported by the GUI in the following sections. JDepend can also provide output in either text or XML format for integration with other tools or reports within your environment.

Once you understand the JDepend metrics, you may want to automate their collection within your normal build and release cycle. Fortunately, there are optional Ant
tasks for doing just that. If you are using Ant 1.5, you can use an XSL stylesheet to transform your JDepend XML output into an HTML report. These reports can be used as a regular part of your quality or metrics program. Turbine and Maven are two projects utilizing this feature.

Understanding JDepend's Metrics Definitions

JDepend computes a series of metrics based on the dependencies between Java packages. These metrics were first identified in Robert Martin's work with C++ (see *Designing Object Oriented C++ Applications Using The Booch Method*) and later extended to work with Java packages by Mike Clark. If the definitions in Table 1 are slightly intimidating, don't worry. We will tie them back into some familiar design and architecture concepts in the next section.

Table 1. JDepends metric definitions

	Metric	Definition
CC	Concrete Classes	The number of concrete classes in this package.
AC	Abstract Classes	The number of abstract classes or interfaces in this package.
Ca	Afferent Couplings	The number of packages that depend on classes in this package. Answers the question "*How will changes to me impact the rest of the project?*"

Ce	Efferent Couplings	The number of other packages that classes in this package depend upon. Answers the question "*How sensitive am I to changes in other packages in the project?*"
A	Abstractness	Ratio (0.0-1.0) of Abstract Classes (and interfaces) in this package. AC/(CC+AC)
I	Instability	Ratio (0.0-1.0) of Efferent Coupling to Total Coupling (Ce/(Ce+Ca)).
D	Distance from Main Sequence	The perpendicular distance of a package from the idealized line A+I=1. Answers the question "*How balanced am I in terms of Abstractness and Instability?*" The range of this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence (balanced) and D=1 indicating a package that is as far from the main sequence as possible (unbalanced).

In many ways, packages represent an ideal unit for managing architectural qualities of the system. Packages represent groups of classes, so the package definition must accommodate the broader purpose of the classes that it represents. A well-defined package architecture allows the system to be partitioned into major subcomponents, which supports the isolation of concerns and the ability to understand and reason about the architecture in manageable chunks.

The second strength of packages is that their dynamic components, package dependencies, are tied to the implementation of the classes contained within the package. Because of this, the dependencies can be determined automatically and are representative of the true state of the architecture at that given point in time.

While package dependencies allow us to reason about the structure and relationships within our architecture, this ability is diminished by *cyclic*
package dependencies. Figure 2 illustrates the ways in which a package can become cyclically dependent.

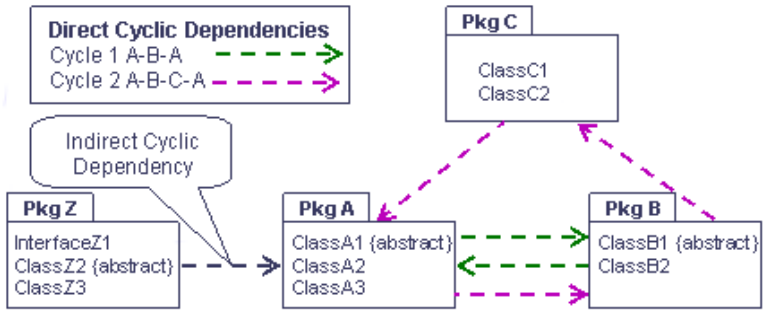

Figure 2. Cyclic dependencies

In the simplest form, two packages, A and B, are cyclically dependent if package A depends on package B and package B depends on package A. I refer to this as a *direct*
cyclic dependency because the packages are directly in the cycle. Note that these *direct* cycles can also span multiple packages: for example, A depends on B, B depends on C, and C depends on A. The second form of cyclic dependency in figure 2 is what I call an *indirect* cyclic dependency. Package Z doesn't directly participate in a *direct* cycle, but because it depends on one (or more) packages that do participate in a *direct* cyclic relationship, it is inherently less stable.

Cyclic dependencies have the following negative consequences for the system:

- Diminish the ability to reason about components of the architecture in isolation.
- Changes impact seemingly unrelated components of the architecture. This makes it difficult to accurately assess and manage the impact of changes to the system.
- Separation of layers. Most architectural approaches recognize the advantages of layered architectures. Cyclic dependencies across layers couple the layers, defeating the purpose of layering.
- Packages cooperating in a cycle must be released as an atomic unit.

Because of the negative consequences of cyclic package dependencies, it is often better to catch them before they make it into your software baseline. The following code sample demonstrates how to write a JUnit test using JDepend to verify that no cyclic dependencies exist for a particular package.

```
package com.xyz.ejb;

import java.io.*;
import java.util.*;
import junit.framework.*;
```

```
import jdepend.framework.*;

public class CycleTest extends TestCase {
..
  /**  Tests that a single package does not contain
    *  any package dependency cycles.
    */
 public void testOnePackageCycle() {

  JDepend _jdepend = new JDepend();

  _jdepend.addDirectory("/projects/ejb/classes");
  _jdepend.addDirectory("/projects/web/classes");

  _jdepend.analyze();

  JavaPackage p = _jdepend.getPackage("com.xyz.ejb");

  assertNotNull(p);
  assertEquals("Cycle exists: " + p.getName(),
               false, p.containsCycle());
 }
..
}
```

The `CycleTest` simply creates a new instance of the `jdepend.framework.JDepend` class, initializes the instance with the directories containing the classes to be analyzed, and then calls the instance's `analyze()` method. After that, you just ask the `JDepend` instance for the `JavaPackage` in which you are interested. Each `JavaPackage` contains all of the metrics we just described. In this case, we are only requiring that the package not include any cyclic dependencies, but you can adjust the test as needed for your project.

Now that we have discussed the negative consequences of cyclic dependencies on package design, we can address to other metrics JDepend provides for assessing package design. You can also think about the other metrics in terms of the graphical representation shown in Figure 3.

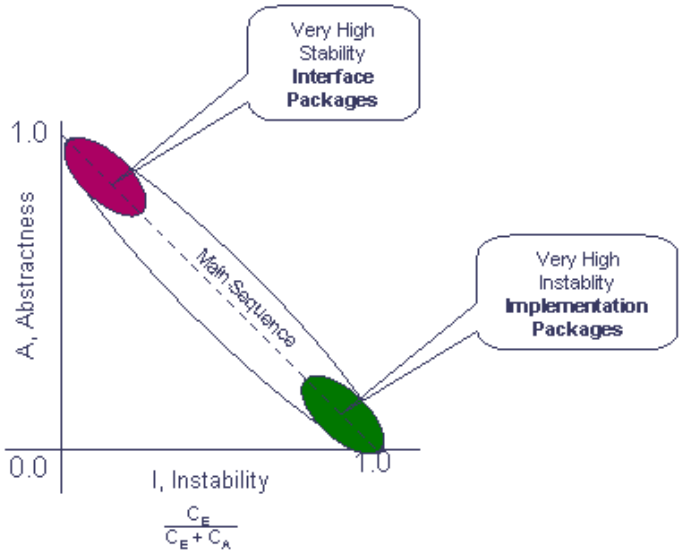

Figure 3. Graphical view of JDepends metrics

From an architectural perspective, there are two primary categories for packages: *Interface* packages and *Implementation* packages.

Interface Packages

In most projects of any size, you typically spend some time identifying and defining the key contracts or interfaces between the major components of your system. This arrangement allows various development groups to proceed in parallel using the interface as an informal contract that will allow them to successfully integrate at a future date.

You will typically capture this contract information as either Java interfaces and/or Java abstract classes that can be grouped, as a cohesive unit, in a Java package. From an OO perspective, you have a package with little or no implementation that is being used throughout the rest of the system. This conforms to the general guidance:

Program to an interface, not to an implementation.

These Interface packages are represented by the ellipse in the upper left hand corner of Figure 3. Because they are only composed of Java interfaces and/or abstract classes, they have a high Abstractness (A). Because they are used by other packages but have few (if any) dependencies on other packages, they tend to be very stable. That is to say, they have low Instability (I).

Implementation Packages

At the other end of the spectrum from Interface packages are Implementation packages. These packages are made up of predominately concrete classes that represent the implementation(s) of the various components of the system. The implementation classes are represented by the ellipse in the lower right hand corner of Figure 3. The key is that the classes in these Implementation packages may depend on all of the other packages in the system, but no other packages should depend on them. Because of this, the implementation is free to change without having these changes ripple through the rest of the system.

Low-abstraction packages should depend upon high-abstraction packages.

This fits well with the concept of introducing new implementations to improve maintainability or performance of one portion of the system without having the change ripple through other unrelated portions of the system.

Main Sequence

It would be nice if everything fit neatly into the categories of pure interface or pure implementation, but the real world involves compromises and trade-offs. The Main Sequence, shown by the dashed line in Figure 3, represents the notion that although the forces of Abstractness and Instability for a package may vary, they should vary proportionally to one another. The ellipses around the Main Sequence are intended to show that the JDepend metrics are generalized, versus absolute measures of package architecture quality. JDepend reports D, which is actually the perpendicular distance from the Main Sequence, to simplify the math. This is shown by the *d1* and *d2* in the figure. There are no absolutes for the value of D, but as its distance from the Main Sequence increases, there is a higher likelihood that the package(s) could benefit from a review or refactoring.

Analyzing the Pet Store

As an example of how you might apply JDepend in a project setting, let's take a look at the results of analyzing the popular Java Pet Store application using JDepend.

I downloaded the 1.3 version of Java Pet Store
from the Sun site and extracted the class files from the .jar files into a common directory, *C:/petstore*. I then ran JDepend using the Swing GUI:

```
java -cp %CLASSPATH% jdepend.swingui.JDepend C:/petstore
```

Figure 4 shows the resulting Afferent Dependencies portion of the JDepend GUI.

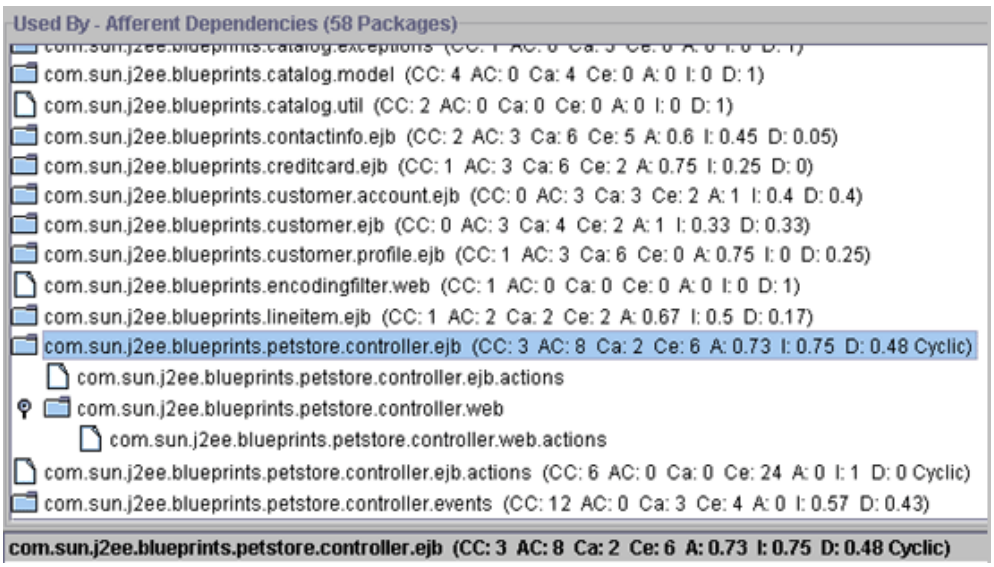

Figure 4. Afferent dependencies for petstore.controller.ejb

Reviewing the results in the GUI shows the dreaded "cyclic" tag on the end of a number of the packages. The `petstore.controller.ejb` package has been expanded to show the packages that depend upon it, `controller.ejb.actions` and `controller.web`. You can also see that `controller.web.actions` is the only package that depends upon `controller.web`. Since we know cyclic dependencies are generally not desirable, let's see if we can identify their causes first.

Cyclic Dependencies

JDepend identifies packages that are cyclically dependent, but it also labels packages that depend on cyclically dependent packages as cyclic. These are the direct and indirect cyclic dependencies we discussed earlier. When refactoring to remove cyclic dependencies, your goal is to remove direct cyclic dependencies. Indirect cyclic dependencies are really just a side effect of the direct cyclic dependencies representing the ripple of the instability through the system. You can use the Swing GUI to identify the type of cyclic dependency your package is participating in.

Referring back to Figure 4, if the `petstore.controller.ejb` package is part of a direct cyclic dependency, then we would see the package names begin to repeat as we drilled down into the dependency structure. Because we do not see a pattern of repeated package dependencies in the afferent window, we know the package is cyclically dependent because it depends on a package that contains a cycle (indirect cyclic dependency). Figure 5 shows the efferent dependencies for the `controller.ejb` package.

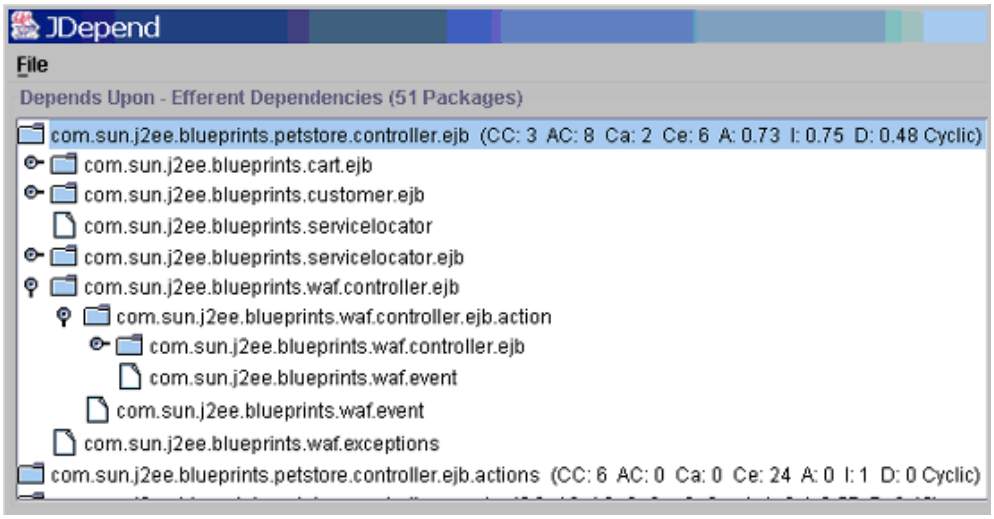

Figure 5. Efferent dependencies for petstore.controller.ejb

Figure 5 shows that `petstore.controller.ejb` depends on six packages:

- `cart.ejb`
- `customer.ejb`
- `servicelocator`
- `servicelocator.ejb`
- `waf.controller.ejb`
- `waf.exceptions`

I've expanded the dependencies so you can see them in the display, but you should also note that the **Ce** metric also tells you there are six packages that this package depends upon. I drilled down further into the `waf.controller.ejb` package to allow you to see that the cycle is caused by dependencies between the `waf.controller.ejb` and `waf.controller.ejb.action` packages. The interface and abstract class composing the `waf.controller.ejb.action` package are shown below.

```
// ...waf.controller.ejb.action.EJBAction
// all packages begin with com.sun.j2ee.blueprints

package waf.controller.ejb.action;
import waf.event.Event;
import waf.event.EventResponse;
import waf.controller.ejb.StateMachine;
import waf.event.EventException;
```

```
public interface EJBAction  {

  public void init(StateMachine urc);
...
}

// ...waf.controller.ejb.action.EJBActionSupport
// all packages begin with com.sun.j2ee.blueprints

package waf.controller.ejb.action;
import waf.controller.ejb.StateMachine;

public abstract class EJBActionSupport
 implements java.io.Serializable, EJBAction {

  protected StateMachine machine = null;

  public void init(StateMachine machine) {
      this.machine = machine;
  }
...
}
```

Note that both of these files meet the general criteria for an Interface package. However, both the interface and the abstract class depend on the concrete class `com.sun.j2ee.blueprints.waf.controller.ejb.StateMachine`. As I mentioned earlier in the article, Interface packages should, if possible, generally avoid depending on concrete Implementation packages. In this case, simply defining an interface, `waf.controller.ejb.action.StateMachineIF`, for the `EJBActionSupport` class and `EJBAction` interface to use in their definitions would correct the cycle. The `StateMachine` class could then implement the `StateMachineIF` and the dependencies would only flow into the `ejb.action` package.

Larger Distance from Main Sequence

Another package that stood out in a quick review of the Pet Store packages is the `servicelocator.ejb` package, which had the following metrics:

```
CC: 1   AC: 0   Ca: 12   Ce: 1   A: 0   I: 0.08   D: 0.92
```

The distance from the main sequence, D, was what brought the package to my attention. Let's look at the story the metrics tell about the package. First, there is only one concrete class in the package (CC = 1) and no abstract classes (AC = 0), so it falls into the general category of an Implementation class. Next, the package is depended upon by 12 other packages (Ca = 12) but only depends on one other package (Ce = 1). Because there are no abstract classes, the Abstractness is 0 (A = AC/(AC+CC)= 0). Because 12 packages depend on this package and it depends on only 1 package, the Instability is 0.08 (I = Ce/(Ce+Ca)).

At this point, you can begin to see several contradictions in the package. The package is heavily depended, so it should have a high degree of abstraction, but it is composed of only one concrete class. The package consists of only one class, so the JDepend metrics are really class metrics in this case. The class is also used by 12 other packages, so changes to the `ServiceLocator`
class will ripple through these packages. The incredible part is that we already know quite a bit about the package and its relationships, even though we have not yet looked at the source. If you are using the JDepend GUI, you can also find out the specific packages this package depends upon and the packages that depend on this package.

From the package name, you may have already guessed that the concrete class contained in the `servicelocator.ejb` package is an implementation of the ServiceLocator pattern from the Core J2EE Patterns book. The `asyncsender.ejb.AsyncSenderEJB` implementation shows how the `ServiceLocator` is used by one of these packages, `asyncsender.ejb`.

```
//asyncsender.ejb.AsyncSenderEJB
// all packages begin with com.sun.j2ee.blueprints

package asyncsender.ejb;

import asyncsender.util.JNDINames;
import servicelocator.ejb.ServiceLocator;
import servicelocator.ServiceLocatorException;

public class AsyncSenderEJB implements SessionBean
{
  private SessionContext sc;
  private Queue q;
```

```
  private QueueConnectionFactory qFactory;
  ....
  public void ejbCreate( ) throws CreateException
  {
    try {
      ServiceLocator serviceLocator =
        new ServiceLocator();

      qFactory =
        serviceLocator.getQueueConnectionFactory(
         JNDINames.QUEUE_CONNECTION_FACTORY);

      q = serviceLocator.getQueue(
        JNDINames.ASYNC_SENDER_QUEUE);

    }catch (ServiceLocatorException sle) {
      throw new EJBException(
        "AsyncSenderEJB.ejbCreate failed", sle);
    }
  }
  ....
}
```

You can see that the `AsyncSenderEJB` class creates and holds a local reference to an instance of the concrete class, `ServiceLocator`. A more flexible approach would be to define a new interface, `servicelocator.ejb.ServiceLocatorIF`, that the `ServiceLocator` class could implement. Because JDepend considers `ServiceLocatorIF` to be an abstract class, this would improve the Abstractness of the `servicelocator.ejb` package from 0 to A = AC/(AC+CC) = 1/(1+1) = 0.5. Points D1(original) and D2(modified) in Figure 6 illustrate the impact of this change on this distance from the main sequence for `servicelocator.ejb` package.

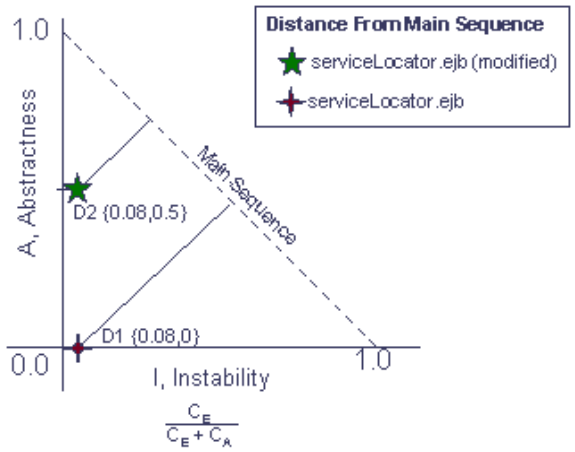

Figure 6. `servicelocator.ejb` package analysis

Note that while this improves the `servicelocator.ejb` package's metrics, it does not address the problems of the classes that use the `ServiceLocator` class. Currently, adding a new type of `ServiceLocator`, say a `CachingServiceLocator`, would require modifying all 12 packages (24 files) that use the `ServiceLocator`. A more flexible approach would be to declare the local variable to be of type `ServiceLocatorIF`. The creation of the correct type of `ServiceLocator` could then be abstracted behind a factory method. This is left as an exercise for the reader.

Some Parting Remarks

There is always the tendency to attempt to identify a metric(s) that can be cast in stone as the ultimate proof of good software or architectural quality. I've had great success utilizing the metrics described in this article, on both Java and C++ projects, to identify architecture and implementation hotspots. I've never found a desirable cyclic dependency; however, I have found cases where the cost to fix the cycle was too high. The distance metric, D, also provides a reasonable approach to organizing classes into cohesive packages, but there is not a magical value for D.

Architectural "ilities" are often difficult to quantify in a concrete and repeatable manner. As your team size grows, it becomes impossible to ensure good design qualities are maintained by simply reviewing all the implementations. As you iterate through the design-implement-refactor cycle, it is relatively easy to introduce undesirable dependencies or erode the cohesion of one or more packages. JDepend provided a series of simple, repeatable metrics that allow you to monitor the evolution of your package architecture as part of your normal development process. The ultimate decision as to what to do is still up to the team, but the key is that you now have a simple, repeatable approach for monitoring the impacts of design and implementation decisions on the architecture.

Glen Wilcox has been developing software in various industries for over 15 years.

Return to ONJava.com.

Copyright © 2007 O'Reilly Media, Inc.