AOP@Work: Design with pointcuts to avoid pattern density

Usability and maintainability in the JUnit Cook's Tour

Wes Isberg
June 14, 2005

In "JUnit: A Cook's Tour," authors Erich Gamma and Kent Beck discuss the design of JUnit. They point out that `TestCase`, like key abstractions in many mature frameworks, has a high pattern density, making it easy to use but hard to change. In this installment of the AOP@Work series, Wes Isberg revisits the Cook's Tour and shows you how using AOP pointcuts rather than object-oriented designs can help you avoid some of the pattern density that makes mature designs hard to change.

View more content in this series

Even the best Java™ programs seem to get old with time. As the design evolves to meet new needs, key objects are loaded with pattern roles until they become hard to use or hard to change and the system has to be refactored or rewritten. Aspect-oriented programming (AOP) offers gentler ways to assemble features and offer services, ways that can minimize interactions and effort to help you extend the life of your design and code.

In this article, I walk through the design presented by Erich Gamma and Kent Beck in "JUnit: A Cook's Tour" (see the Resources section). For each Java pattern they present, I propose an AspectJ alternative and evaluate how it satisfies some of these canonical design goals:

- **Functionality**: How powerful or useful are the services presented?
- **Usability**: How easy is it for a client to get services?
- **Extensibility**: How easy is it to extend or adapt when the program changes?
- **(De)Composability**: How well does it play with others?
- **Protection**: How secure is an API against runtime or cascading errors?
- **Understandability**: Does the code make sense?

At each step in their design, Gamma and Beck were faced with choices pitting, for example, usability against maintainability or understandability against composability. With each choice, they took the path of simplicity and usability, even when it meant giving up on a secondary concern. As a result, their design makes it very easy to write unit tests. However, I would still like to ask, would they have been able to avoid some of these design tradeoffs had they used AOP?
To ask this, I have to be more demanding than is reasonable. JUnit does what it does very well, and the design makes tradeoffs that many developers have learned to accept as normal. To see if AOP can do better, I have to ask, for example, whether I can add more features and make it more usable to handle clients who want more services without having to comply even with JUnit's modest demands. I do this not to fix JUnit, but to avoid giving up secondary design goals to achieve primary ones.

I use AspectJ for all examples in this article, but the examples could work with other AOP approaches and should make sense even to those new to AspectJ. (Indeed, it's probably more helpful if you've read the Cook's Tour and know design patterns than if you've used AspectJ or JUnit.) To download the source code for this article, click the Code icon (or go to Download) at the top or bottom of the page.

Command or Hypothesis?

Here's the starting point for Gamma and Beck in "JUnit: A Cook's Tour":

Developers often have tests cases in mind, but they realize them in many different ways: print statements, debugger expression, test scripts. If we want to make manipulating tests easy, we have to make them objects.

To make tests into objects, they used the Command pattern, which encapsulates "a request as an object, thereby letting you [...] queue or log requests." What could be more straightforward?

Given their focus on usability, it seems odd for Gamma and Beck to see that developers write tests in different ways, but then insist that developers write tests in only one way: encapsulated as an object. Why do they do this? To make it easier to work with tests. There's the rub: to benefit from the service, you have to comply with the form.

That trade-off affects both the shape and evolution of a design. You target a certain client with a blend of usability and power and build your system. If the client changes, you add layers or change the blend, each time working within and around the system you already built. With luck, your system has enough degrees of freedom for the process to converge on a solution for your client. Gamma and Beck frame this convergence in terms of pattern density:
Once you discover what problem you are really solving, then you can begin to "compress" the solution, leading to a denser and denser field of patterns where they provide leverage.

**Pattern density by design**

Having identified a test case as the key abstraction and encapsulated it using `Command`, the Cook's Tour proceeds to identify new requirements and add new features to the objects representing the key abstractions. The result is nicely summarized in the following storyboard:

**Figure 1. JUnit pattern storyboard**

![JUnit pattern storyboard](image)

Gamma and Beck are following (I should say leading) what is by now standard operating procedure for design: find the key abstractions, encapsulate them in objects, and add patterns to arrange for various roles they play and services they offer. Unfortunately, this is a recipe for pattern density. The key abstractions accrue responsibilities and relationships until, like parents stuck in middle age, they can only follow their well-worn patterns. (And if their needs still outrun what they can do, a crisis ensues.)

**Given a test case...**

AOP offers another way to specify an abstraction: as a pointcut specifying a join point. A **join point** is a point in the execution of a program where you can usefully join behavior. The kinds of join points vary depending on your flavor of AOP, but all should be stable across insignificant program changes and easy to specify meaningfully. You use a **pointcut** to specify join points of a program, and **advice** to specify the behavior to join. Advice is a way to say "When X, do Y."

The Command pattern says, "I don't care what the code to run is; just put in this method." It requires you to put the code in the command method of a command class -- for JUnit, in the `runTest()` method of a `Test` such as `TestCase`:

```java
public class MainTest extends TestCase {
    public void runTest(...) {...}
}
```
By contrast, a pointcut says, "Let some join point be a test case." It requires only that a test case be some join point. Instead of making you put the code in a particular method in a particular class, you need only specify a test case with a pointcut:

```java
pointcut testCase() : ... ;
```

For example, you could define as test cases `Runnable.run()`, main methods, and of course JUnit tests:

```java
pointcut testCase() : execution(void Runnable+.run());
pointcut testCase() : execution(static void main(String[]));
pointcut testCase() : execution(public void Test+.run(TestResult));
pointcut testCase() : execution(public void TestCase+.test*());
```

**Usability of pointcuts**

The usability of a pointcut is hard to beat. In this case, as long as the test can be picked out by a pointcut, it can be treated as a test case -- even if it was not written as a test. If you can offer services as advice rather than through APIs, you minimize the work developers have to do for those services.

With AOP, you can provide services without any effort by the developer. This gives you new kinds of API clients to consider: those you can help without their knowledge, or who simply rely on the service. With normal APIs, there is an explicit contract between client and provider and a specific time they are called. With AOP, it's more like the way people rely on the government; whether calling the police, registering at the DMV, or just eating or banking, people rely (more or less consciously) upon regulation to operate (more or less explicitly) at well-defined points.

With AOP in the picture, usability becomes a broader continuum, from API contracts through container-based programming models to the many forms of AOP. The question of usability shifts from how much work the service interface imposes on the the client to how much knowledge and choice about the service clients want or need (whether they are drivers, crooks, or job applicants).

**Reusability**

Like a method, a pointcut can be declared abstract; you can use it in advice, but let subaspects declare it concretely. Typically, an abstract pointcut specifies not a specific time or location ("Tuesday at 1600 Pennsylvania Avenue") but a general happening of interest to many ("the election"). Then you can say things true of any such happening ("During the election, the news agencies..." or "After the election, the winner...") and your users can specify the when, who, and where of a given election. Specifying a test case as an abstract pointcut, I'm betting not only that many features for a test harness can be expressed in the form "When X, do Y," but also that I can write most of the do Y without knowing too many details about the When X.

How can using advice to implement features avoid the perils of pattern density? When I add new features to a class, each new member can see the other visible members, so the theoretical complexity increases. By contrast, AspectJ minimizes interactions between advice. Two pieces of advice at a join point are not visible to each other, and they only bind the join point context...
variables they declare. If one advice does affect another and needs to be ordered, I can specify their relative precedence, rather than having to know about all the advice and specify a complete order. Each advice uses as little information about the join point as possible and discloses about itself only what's necessary for type-safety, exception-checking, and the like. (AspectJ is nearly unique among AOP technologies in supporting encapsulation at this level.) With a minimum of interactions, complexity should grow less when advice are added to a join point than it does when members are added to a class.

For the rest of the Cook's Tour, I implement features with the `testCase()` pointcut as Gamma and Beck add them to `TestCase`. At each step, I try to avoid the trade-offs they had to make by assessing whether order matters at the join point, by avoiding assumptions about the context at the join point, and by supporting as many kinds of API clients as is sensible.

**Template Method or around advice?**

Having used Command to encapsulate the test code, Gamma and Beck recognize a common flow to tests that use some common data fixture: "set up a test fixture, run some code against the fixture, check some results, and then clean up the fixture." To encapsulate this, they use the Template Method pattern:

```
Quoting from the intent, "Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."
```

In JUnit, developers use `setUp()` and `cleanUp()` to manage data for a `TestCase`. The JUnit harness is responsible for invoking the methods before and after each test case runs; `TestCase` does this using the template method `runBare()`:

```
public void runBare() throws Throwable {
    setUp();
    try {
        // run the test method
        runTest();
    } finally {
        tearDown();
    }
}
```

In AspectJ, when code needs to run before and after a join point, you can use the combination of `before` and `after` advice; or a single piece of `around` advice, as I've done here:

```
/** around each test case, do setup and cleanup */
Object around() : testCase() {
    setup(thisJoinPoint);
    try {
        // continue running the test case join point
        return proceed();
    } finally {
        cleanup(thisJoinPoint);
    }
}
protected void setup(JoinPoint jp) {}
protected void cleanup(JoinPoint jp) {}
```
Advice like this offers three degrees of freedom:

- It can work on any join point supporting *around* advice.
- It can work on any kind of test, since it makes no assumptions about it.
- By putting the fixture setup/cleanup code in methods that can be overridden or implemented by delegation, it can adapt to different ways of managing fixtures presented by different kinds of test objects. Some might manage their own data, as `TestCase` does; others might benefit from dependency inversion, where their configuration is established extrinsically. However, these methods use `JoinPoint`, which provides as `Object` any context available at a join point (possibly including `this` object, the `target` object, and any arguments). Using `JoinPoint` will entail a downcast from `Object` to any actual type, trading generality for type-safety. (Below I will suggest a way to get type-safety without losing generality.)

This advice offers the same guarantees as Template Method but without the constraints of a Java implementation. In JUnit, `TestCase` has to take control of the command method to implement the template method, and then delegate to another method for the real test, creating a `TestCase`-specific protocol for the command code. As a result, while Command makes the test easy to manipulate, the command contract actually varies for the developer from `Test` to `TestCase`, making API responsibilities harder to understand.

**Collecting Parameter or ThreadLocal?**

The Cook’s Tour continues on its peripatetic way: "If a `TestCase` runs in a forest, does anyone care about the result?" Of course, Gamma and Beck reply: You need to record failures and summarize success. To do that, they used the Collecting Parameter pattern:

> When you need to collect results over several methods, you should add a parameter to the method and pass an object that will collect the results for you.

JUnit encapsulates result handling in a single `TestResult`. This is where subscribers can find results from all tests and where the harness can manage result-collecting concerns. To do the actual collecting, the Template Method `TestResult.runProtected(..)` brackets test execution with `start` and `end` housekeeping calls and interprets exceptions thrown as negative test results.

**Composability**

Now that there are $N>1$ patterns, what is the interaction between pattern implementations? When objects play well with each other, they are said to be *composable*. Similarly, pattern implementations can conflict directly (e.g., if two require a different superclass), coexist but not interact, or coexist but interact, in more or less fruitful ways.

In JUnit, the interplay between the fixture concern and the result-collecting concern entails a call-sequence protocol shared between `TestCase` and `TestResult`, as shown here:

```
Test.runTest(TestResult) calls...
TestResult.run(TestCase) calls...
   TestResult.runProtected(Test, Protectable) calls...
   Protectable.protect() calls...
   TestCase.runBare() calls...
   Test.runTest() ...
   (TestCase.runTest() invokes test method...)
```
This shows how pattern density makes code hard to change. If you wanted to change either the fixture template method or the collecting parameter, you would have to change both, in both TestResult and TestCase (or your subclasses). Further, because the fixture setUp() and cleanUp() methods run in the protected context of result handling, this call sequence encodes a design decision: Any exceptions thrown in fixture code are treated as test errors. If you wanted to report fixture failures separately, you would have to change not only both components, but also the way they call each other. Can AspectJ do better?

In AspectJ, you can use advice to provide the same guarantees but avoid locking down the order:

```java
/** Record test start and end, failure or error */
void around(): testCase() {
    startTest(thisJoinPoint);
    try {
        proceed();
    } catch (Error e) {
        error(thisJoinPoint, e);
    } catch (Exception e) {
        failure(thisJoinPoint, e);
    } catch (Error e) {
        error(thisJoinPoint, e);
    }
}
```

Like the fixture-handling advice above, this could work for any kind of test or result collector, but implementing the methods would involve downcasting. I'll fix that later. For now, how does this advice interact with the fixture advice? It depends on which runs first.

**Who's up first?**

In JUnit, the template methods for result gathering and fixture management had to be placed (forevermore?) in a particular call sequence. In AspectJ, lots of advice can run at a join point without knowing anything about other advice at the join point. In cases where they do not interact, you can (and should) ignore the order in which they run. However, if you know that one might affect the other, you can control how they run using precedence. In this case, if you give the result-handling advice more precedence, when the join point runs, the result-handling advice starts before the fixture-handling advice, calls proceed(...) to continue, and gets control back when that is done. Here's how it looks at run time:

```
# start running the join point
start result-handling around advice; proceed(...) invokes..
    start fixture-handling around advice; proceed(...) invokes..
        run underlying test case join point
    finish fixture-handling around advice
finish result-handling around advice
# finish running the join point
```

**Talking join points**

When talking about how a join point runs, consider it like a program stack (but don't expect any stack frames). The (whole) join point starts before the first advice starts and finishes after the last advice ends. The "underlying" or "original" join point is the code being advised; it may in turn run other (nested) join points. Advice runs in precedence order before and after the underlying join point. When around advice calls proceed(...), it "continues the join point,“
invoking any less-precedent advice and the underlying join point. Some advice might not run if the pointcut does not match at that time (e.g., if logging is disabled). So when developers say, "The join point ran..." they mean the entire join point, advice and all (even if the advice or the underlying join point didn't run because pointcuts didn't match, proceed(...) wasn't called, or an exception was thrown).

If necessary, you can control precedence of the two advice explicitly, whether the advice are in the same or different aspects, and even from a third aspect. Here, because the order constitutes a design decision about whether fixture errors are reported as test errors, you would want to set the precedence explicitly. I could use a separate aspect to state the policy about fixture errors:

```java
aspect ReportingFixtureErrors {
    // fixture errors reported by result-handling
    declare precedence: ResultHandling+, FixtureHandling+;
}
```

The two Handling aspects don't need to know about each other, unlike the two JUnit classes TestResult and TestCase, which must agree between themselves who runs the command first. To change the design later, I need only change the ReportingFixtureErrors.

### Usability of Collecting Parameter

Most JUnit test developers don't use TestResult directly; that would mean passing it as a parameter in every method in the call chain, which Gamma and Beck call "signature pollution." Instead, they offer JUnit assertions to both signal the failure and unwind the test.

TestCase extends Assert, which defines a number of useful static assert{something}(...) methods for checking and logging failures. When the assertions fail, the methods throw AssertionFailedError, which TestResult catches and interprets in the result-handling fixture template method. In this way, JUnit neatly sidesteps the API user's problem of passing around collecting parameters and enables users to be oblivious to the requirements of TestResult. JUnit bundles the concern of reporting results with the service of verification and logging.

### Bundling

Bundling makes it harder for users to pick services they want. Using Assert.assert{something}(...) ties TestCase to TestResult more deeply and hides the flexibility of a collecting parameter. It forces fast-fail semantics on tests, even though some tests might want to continue past a verification failure. To report results directly, JUnit tests could implement Test, but then they would lose the other features of TestCase (pluggable selector, fixture handling, rerunning test cases, etc.).

This is another cost of pattern density: API users are often forced to accept or reject a whole package. Further, while it can be convenient to bundle concerns, sometimes it can reduce reusability. For example, many class or method invariants are first written as JUnit assertions; these invariant checks could be reused for production diagnostics if it did not automatically trigger exceptions.

As shown above, AspectJ can support result handling for JUnit-style assertions; can it at the same time support API users who want the flexibility of using the result collector directly and deciding separately when to unwind the test? Even if they define their own result collector and report interim
results? I think so. There are four parts to a solution: (1) support factories for result collectors; (2) make the result collector available to components without polluting method signatures; (3) enable tests to unwind after reporting directly to the result collector; and (4) ensure that exceptions thrown are reported properly. These were hard to do when the Cook’s Tour was written, but newer Java APIs and AspectJ make them easier now.

**ThreadLocal collector**

To make the result collector available to all components and to implement a factory, I use a public static method to get a thread-local result collector. Here’s a skeleton `TestContext` result collector:

```java
public class TestContext {
    static final ThreadLocal<TestContext> TEST_CONTEXT = new ThreadLocal<TestContext>();

    /** Clients call this to get test context */
    public static TestContext getTestContext(Object test) {
        ...
    }

    ...
}
```

The method `getTestContext(Object test)` can support different associations between the result collector and the tests (per test, per suite, per thread, per VM), but subtypes of `TestContext` will require a downcast and other types are not supported.

**Unwinding the test**

Throwing an exception not only unwinds the test but also signals an error. If test clients directly signals the error using `getTestContext(..)`, they need to unwind the test without further error reporting. To do that, declare a a special exception class that indicates the result has been signalled already. The API-contract way is to define a class known to both the throwing client and the catching harness. To hide type details from the client, declare a method that returns the exception for users to throw, as follows:

```java
public class TestContext {
    ...
    public Error safeUnwind() {
        return new ResultReported();
    }

    private static class ResultReported extends Error {}
}
```

A test then throws whatever exception the particular `TestContext` defines:

```java
public void testClient() {
    ...
    TestContext tc = TestContext.getTestContext(this);
    tc.addFailure(...);
    ...
    throw tc.safeUnwind(); // could be any Error
}
```
This does bind the test to TestContext, but safeUnwind() is only used by tests that do their own result reporting.

Ensuring exceptions are reported

Below is the advice that collects results for TestContext. It is general enough that it can be used for different test cases and different subtypes of TestContext:

```java
/** Record for each test start and end or exception */
void around() : testCase() {
  ITest test = wrap(getTest(thisJoinPoint));
  TestContext testContext = TestContext.getTestContext(test);
  testContext.startTest(test);
  try {
    proceed();
    testContext.endTest(test);
  } catch (ResultReported thrown) {
    testContext.checkReported(test);
  } catch (Error thrown) {
    testContext.testError(test, null, thrown);
  } catch (Throwable thrown) {
    testContext.testFailure(test, null, thrown);
  }
}

protected abstract Object getTest(JoinPoint jp);
```

Because this advice enforces TestContext invariants, I would nest the aspect inside TestContext. To enable harness developers to specify different test cases, both the pointcut and method are abstract. For example, here is how I could adapt this to TestCase:

```java
aspect ManagingJUnitContext
  extends TestContext.ManagingTestResults {

  public pointcut testCase() : within(testing.junit..*)
    && execution(public !static void TestCase+.test*());

  protected Object getTest(JoinPoint jp) {
    assert jp.getTarget() instanceof TestCase;
    return jp.getTarget();
  }
}
```

I constrained the solution in one important place: The around advice declares it returns void. If I declared that the advice returns object, I could use the advice at any join point. But because I am catching the exceptions, I am returning normally, and I would have to know what object to return. I could return null and cross my fingers, but I prefer to flag the issue to any subaspects rather than have the issue surface as a run time NullPointerException.

While declaring void limits the reach of the testCase() pointcut, it reduces complexity and increases protection. Advice in AspectJ has the same type-safety and exception checking that methods do in the Java language. Advice can declare that it throws a checked exception, and AspectJ will signal an error if the pointcut picks out any join point that does not throw that exception. Similarly, around advice can declare a return value ("void" above), which requires any join point have the same return value. Finally, if I do bind specific types to avoid downcasting (e.g.,
using `this(...)` which I will show later), I have to be able to find those at the join point. These limitations ensure that AspectJ advice enjoys the same build-time safety as a Java method (unlike approaches to AOP based on reflection or proxies).

With these limitations, I was able to support result collecting by clients both oblivious and controlling, without relying on them to enforce the invariants. The solution is extensible both to new kinds of clients and to new kinds of result collectors and what interactions there are have been localized to the `TestContext` class and its subtypes.

**Adapter, Pluggable Selector, or Configuration?**

The Cook's Tour presents `Pluggable Selector` as an alternative to the "class bloat" caused by creating a subclass for each new test case. As the authors put it:

> The idea is to use a single class which can be parameterized to perform different logic without requiring subclassing [...] Pluggable Selector stores a [...] method selector in an instance variable.

`TestCase` thus plays the role of `Adapter` converting `Test.run(TestResult)` to `TestCase.test...()` using the Pluggable Selector pattern with the name field as a method selector. The `TestCase.runTest()` method reflectively invokes the method corresponding to the name field. This convention enables developers to add a test case just by adding a method.

This is easy for the JUnit test developer to use but hard for a harness developer to change or extend. To satisfy `runTest()`, the parameter to the constructor `TestCase(String name)` must be the name of a public instance method with no parameters. It turns out that `TestSuite` implements this protocol, so if you wanted to change the reflective invocation in `TestCase.runTest()`, you would have to change `TestSuite.addTestSuite(Class)`, or vice-versa. To write data- or specification-driven tests based on `TestCase`, you have to create a separate suite for each configuration, encode the configuration in the suite name, and configure each test after it has been defined by `TestSuite`.

**Configuring join points**

Can AspectJ go one step further than selection to handle test configuration? There are two ways to approach configuring tests at a join point.

First, you can configure the join point directly by varying some context available at the join point, like method arguments or the executing object itself. A simple example for `main(String[])` method executions would be to rerun the join point with different `String[]` arrays to generate a number of tests. A more complex example combines two kinds of variants for the context at the join point. Here's advice to see whether a test works on all printers in both color and monochrome:
```java
void around(Printer printer) : testCase() && context(printer) {
    // for all known printers...
    for (Printer p : Printer.findPrinters()) {
        // try both mono and color...
        p.setMode(Printer.MONOCHROME);
        proceed(p);
        p.setMode(Printer.COLOR);
        proceed(p);
    }
    // also try the original printer, in mono and color
    printer.setMode(Printer.MONOCHROME);
    proceed(printer);
    printer.setMode(Printer.COLOR);
    proceed(printer);
}
```

While this code is specific to a `Printer`, it is oblivious to whether the test is for printing or initialization, or to whether `Printer` is the target of a method call or a parameter to a method. So even where advice requires some specific type, it can be more or less unconcerned with where the reference comes from; here the advice delegates to the subaspect defining the pointcuts both which join point and how to get the context.

The second (more general) way to configure tests is to use APIs on the test component. The `Printer` example showed how to set the mode explicitly. For generality, you could support a generic adapter interface, `IConfigurable`, as shown below:

```java
public abstract aspect Configuration {
    protected abstract pointcut configuring(IConfigurable s);

    public interface IConfigurable {
        Iterator getConfigurations();
        void configure(Object input);
    }

    void around(IConfigurable me) : configuring(me) {
        Iterator iter = me.getConfigurations();
        while (iter.hasNext()) {
            me.configure(iter.next());
            proceed(me);
        }
    }
}
```

This advice would only run if some context were `IConfigurable`, but when it runs, it can run the underlying join point many times.

How does this interact with other kinds of tests at the join point, with other advice at the join point, and with any code that runs the join point? As for tests, if the test is not `IConfigurable`, the advice doesn't run. No conflict there.

As for other advice, assuming you define `configuring()` as `testCase()` and include your other advice, because this effectively creates many tests, the result and fixture advice should have lower precedence so they can manage and report distinct configurations and results. Further, the configuration should somehow be embodied in the test identification used by the result collector.
to report results; this is the responsibility of the component that knows the test is configurable and identifiable (more on this point in the next section).

As for code running the join point, unlike the usual around advice, this one calls proceed(...) once for each configuration, so the underlying join point can run many times. In this case, what result should be returned from the advice? As in the result-handling advice, I’m not sure except in the case of void, so I restrict the advice to return void to signal the issue to the test developer writing the pointcut.

**Make what you need**

If I am a harness developer trying to adapt to a test, it seems like I would be adding to the "pattern density" of my test if I had to implement IConfigurable in the test class. To avoid that, in AspectJ you can declare members and parents of other types, including default implementations for interfaces, as long as any definition preserves binary compatibility. Using inter-type declarations increases the type-safety of advice by making it easier to avoid downcasting from Object.

Does it increase the complexity of the target type like other members? Public members declared on other types are visible so they can add to the theoretical complexity of the target type. However, you can also declare the members of the other type as private to the aspect, so only the aspect can use them. That gives you a way to assemble composite objects without the usual collisions and interactions possible if all members were declared in the class itself.

The following code shows an example adapting Run to IConfigurable using the init(String) method:

```java
public class Run {
    public void walk() { ... }
    public void init(String arg) { ... }
}

public aspect RunConfiguration extends Configuration {
    protected pointcut configuring(IConfigurable s) :
        execution(void Run+.walk()) && target(s);
    declare parents : Run implements IConfigurable;

    /** Implement IConfigurable.getConfigurations() */
    public Iterator Run.getConfigurations() {
        Object[] configs = mockConfigurations();
        return Arrays.asList(configs).iterator();
    }

    /** Implement IConfigurable.configure(Object next) */
    public void Run.configure(Object config) {
        // hmm - downcast from mockConfigurations() element
        String[] inputs = (String[]) config;
        for (String input: inputs) {
            init(input);
        }
    }
}

static String[][] mockConfigurations() {
```
Test identifiers
A test identifier can be shared by result reports, selection or configuration, and the underlying test itself. In some systems, it need only tell the user which test was run; in other systems, it needs to be a unique and consistent key across runs, to check which failing tests passed (bugs fixed) and which passing tests failed (regressions). JUnit only offers a representation and gets around the need to share by using `String Object.toString()` to obtain a String representation. An AspectJ harness could make the same assumptions, but it also could augment any test objects with something like `IConfigurable` above, to calculate and store the identifier for a given type of test according to the system requirements. The "same" test could be configured with different identifiers depending on the requirements (e.g., for diagnostics or regression testing), reducing the conflicts possible with pattern density in the Java language. While configuration was local to the aspect and the component being configured (and could thus be private), the identifier could be visible to many concerns and should thus be represented as a public interface.

Composite or Recursion?
The Cook's Tour recognizes that a harness must run lots of tests -- "suites of suites of suites of tests." The Composite pattern meets this need nicely:

To quote its intent "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly."
The Composite pattern introduces the three participants: Component, Composite, and Leaf. Component declares the interface we want to use to interact with our tests. Composite implements this interface and maintains a collection of tests. Leaf represents a test case in a composition that conforms to the Component interface.

This brings the JUnit design full circle, because the `Test.runTest(...)` Command interface is the Component interface implemented by the Leaf `TestCase Composite TestSuite`.

Maintainability
The Cook's Tour points out "how the complexity of the picture jumps when we apply Composite." In this pattern, the node and leaf roles are superimposed on existing components, and both need to know their responsibility in implementing the component interface. A call protocol is defined between them and implemented by the nodes, which also collect the children. This means the nodes know about the children and the harness knows about the nodes.

In JUnit, `TestSuite` (already) knows a lot about `TestCase`, and JUnit test runners assume that they generate a suite by loading a suite class. As you saw with configuration, supporting configurable test involves managing test suite generation. Composite drives up pattern density.

The Composite pattern can be implemented in AspectJ using inter-type declarations as shown above with configuration. In AspectJ, all members can be declared in a single aspect rather
than scattered through the existing classes. This makes it easier to see that the roles are not polluted with concerns from the existing classes, and it’s easier to understand when looking at the implementation that it is a pattern (and not just another member of the class). Finally, Composite is one of the patterns that can be implemented with an abstract aspect using tag interfaces to specify the classes playing the roles. This means you can write a reusable pattern implementation. (For more information on AspectJ implementations of design patterns, see Nicholas Lesiecki’s “Enhance design patterns with AspectJ” in the Resources section.)

Recursion

Can AspectJ solve the original need without resorting to the Composite pattern? AspectJ offers many ways to run multiple tests. The configuration example above suggests one way: associate a list of children with a test, and use advice to recursively run the components on join points picked out by a pointcut `recursing()`. This pointcut specifies the composite operation that should be recursive:

```java
// in abstract aspect AComposite
/** tag interface for subaspects to declare */
public interface IComposite {}
/** pointcut for subaspects to declare */
protected abstract pointcut recursing(IComposite c);
/** composites have children */
public ArrayList<IComposite> IComposite.children = new ArrayList<IComposite>();
/** when recursing, go through all subtree targets */
void around(IComposite c) : recursing(c) {
    // recurse...
}
```

Here’s how you might apply the aspect to `Run`:

```java
public aspect CompositeRun extends AComposite {
    declare parents : Run implements IComposite;
    public pointcut recursing(IComposite c) :
        execution(void Run+.walk()) && target(c);
}
```

Encapsulating join points as objects

Recurse at a join point? Here’s where things get interesting. In AspectJ `around` advice, you can run the rest of the join point using `proceed(...)`. To do this recursively, you close over the rest of the join point by encapsulating the `proceed(...)` call in an anonymous class. To pass this to a recursive method, the anonymous class should extend a wrapper type known to the method. For example, below I define the `IClosure` wrapper interface, wrap `proceed(...)` in the `around` advice, and pass the result to the `recurse(...)` method:

```java
// in aspect AComposite...
```
/** used only when recursing here */
public interface IClosure {
    public void runNext(IComposite next);
}

/** when recursing, go through all subtree targets */
void around(IComposite c) : recursing(c) {
    recurseTop(c, new IClosure() {
        // define a closure to invoke below
        public void runNext(IComposite next) {
            proceed(next);
        }
    });
}

/** For clients to find top of recursion. */
void recurseTop(IComposite targ, IClosure closure) {
    recurse(targ, closure);
}

/** Invoke targ or recurse through targ's children. */
void recurse(IComposite targ, IClosure closure) {
    List children = (null == targ?null:targ.children);
    if ((null == children) || children.isEmpty()) {
        // assume no children means leaf to run
        closure.runNext(targ);
    } else {
        // assume children mean not a leaf to run
        for (Object next: children) {
            recurse((IComposite) next, closure);
        }
    }
}

Using IClosure combines the benefits of the Command pattern with those of advice using proceed(...). Like Command, it can be passed around to be run or rerun optionally with newly-specified parameters. Like proceed(...), this hides the details of other context available at the join point, of any other less-precedent advice, and of the underlying join point itself. It is as general as a join point, safer than advice (because the context is more hidden), and as reusable as Command. Because it imposes no requirements on the target types, it is more composable than Command.

Don't be surprised if closing over proceed(...) takes getting used to; to many Java developers, it's just plain weird. And if you invoke the IClosure object after the join point completes, results may vary.

**Usability**

The RunComposite aspect applies this composite solution to the Run class just by tagging the class with the IComposite interface and defining the recursing() pointcut. However, to assemble components in a tree entails adding children; that means some assembler component has to know that Run is a IComposite with children. Here are the components and their relationships:

Assembler, knows about...
- Run component, and
- CompositeRun concrete aspect, who knows about...
  - Run component, and
  - AComposite abstract aspect
You might try to make the CompositeRun aspect also responsible for finding children for each run (likely in combination with configuration), but using a separate assembler means you don't muddy the compositeness-of-Run (which is constant for all runs) with the particular application of the Run composite (which varies with how to associate children with a particular Run subclass). The rule for object-oriented dependencies is to depend in the direction of stability; in particular, (more variable) concrete elements should depend if at all on (more stable) abstract ones. In light of that rule, the above dependencies look right.

**Composability**

Like configuration, composite advice (when applied to test cases) should take precedence over fixtures and result reporting. If configuration affects test identity, then composite should take precedence over that, too. Given those constraints, here's one ordering:

<table>
<thead>
<tr>
<th>Composition</th>
<th>Configuration</th>
<th>Context</th>
<th>Fixture</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td># recursion</td>
<td># defining test, identity</td>
<td># reporting results</td>
<td># managing test data</td>
<td># underlying test</td>
</tr>
</tbody>
</table>

**Pointcut as a design abstraction**

That wraps up my look at the JUnit Cook’s Tour. All of the aspect-oriented design solutions I’ve discussed are available complete in the code bundle for this article. The solutions have the following characteristics:

- They rely on a pointcut rather than a type, either assuming nothing or deferring a context specification to a concrete subaspect.
- They stand alone and can be used alone.
- They can be reused, many times in the same system.
- They can work together; in some cases you will want to define their relative precedence.
- They require no changes on the part of the client.
- They do more than JUnit could with less effort for the client.

For a given Java pattern, AspectJ provides many ways to do the same thing, sometimes with a simple idiom. I took the approach here of using a pointcut and assuming the least possible in reusable solutions, mainly to demonstrate how AspectJ is designed to encapsulate advice and the join point to minimize interactions, making it easy to scale up behavior at a join point. In some cases, it might be clearer to use a concrete (non-reusable) aspect, to combine features into one aspect, or to use inter-type declarations to implement the corresponding Java patterns. However, these solutions demonstrate techniques for minimizing interactions at a join point, to make it easier to use pointcuts as first-class design abstractions.

Pointcuts are just the first step in a design approach that minimizes entangling assumptions. Try to really leverage join points where objects might not be necessary. Where objects are necessary, try to use inter-type declarations in aspects to compose the objects, so different (pattern) roles remain distinct even when defined on one class. As with object-oriented programming, try to keep different components from knowing about each other. If they must know about each other, the
concrete ones should know about the abstract ones, and the assemblers should know about the parts. When they know about each other, the relationships should be narrow, explicit, stable, and enforceable.

**Full-speed AOP**

AspectJ 1.0 was released over three years ago (!). Most developers have seen or tried the entry applications of AspectJ, which modularize a crosscutting concern like tracing. But some developers are going further, attempting what I would categorize as full-speed AOP:

- Designs fail as often as they succeed.
- Crosscutting specifications like pointcuts are reused or reusable.
- Aspects can have many elements that rely on each other.
- Aspects are used to invert dependencies and decouple code.
- Aspects are used to connect components or subsystems.
- Aspects are packaged as reusable binary libraries.
- Many aspects are in the system. Some are oblivious to others and some rely on others.
- While an aspect might be unpluggable, when plugged-in it adds essential functionality or structure.
- Base code is refactored to create a better join point model.

What holds some developers back from going full-speed? People seem to reach a plateau after initially hearing about AOP or learning the basics. One thought-trap goes like this:

AOP modularizes crosscutting concerns, so I should look for crosscutting concerns in my code. I've exhausted all the tracing, synchronization, etc., I need, so there's no more to do.

This trap is similar to thinking solely in terms of "is-a" and "has-a" in the early days of object-oriented programming. By looking for a single concern (even if it is crosscutting), you miss the relationships and protocols that, when normalized as patterns, form the backbone of code practice.

Another thought-trap goes like this:

AOP modularizes crosscutting concerns, so I should look for code that is scattered and tangled in my code. It all seems well-localized, so I don't need AOP.

While it is true that scattering and tangling are indicia of unmodularized crosscutting concerns, there are many useful applications of AOP that don't involve gathering scattered code or detangling complex methods or objects.

Finally, the hardest thought-trap to escape:

AOP augments object-oriented programming with new language facilities, so it should work to solve problems that object-oriented programming can't. Object-oriented programming solves all my problems, so I don't need AOP.
In this article, I did not look for crosscutting concerns, and most of what I reimplemented was, to current thinking, well-modularized. While the code I presented was not a complete success (especially measured against JUnit), the point wasn't just to show that it could be done or prove that the code can be better localized, but the point was to question whether it's necessary to endure the design trade-offs object-oriented developers have come to accept. I believe that if you can avoid having to give up secondary design goals to meet your primary goals, you can avoid writing code that becomes hard to use or hard to change.

Conclusion

Revisiting "JUnit: A Cook's Tour" was a good way to develop a better understanding of the ways AspectJ can minimize and control interaction at a join point, which is key to using pointcuts effectively in your design. Pattern density, which can make mature object-oriented frameworks hard to change, is a natural result of the way object-oriented developers design systems. The solutions presented here, by using a pointcut instead of an object, avoiding interactions where possible, and minimizing them otherwise, avoid the inflexibility of JUnit and show how you can do the same in your own designs. By avoiding design trade-off's developers have grown to accept, these solutions show that AOP can be useful even when the code is already thought to be well-modularized. I hope you are encouraged to try AOP in more applications, at full speed.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>j-aopwork7code.zip</td>
<td>41 KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2005  

**Trademarks**  
(www.ibm.com/developerworks/ibm/trademarks/)