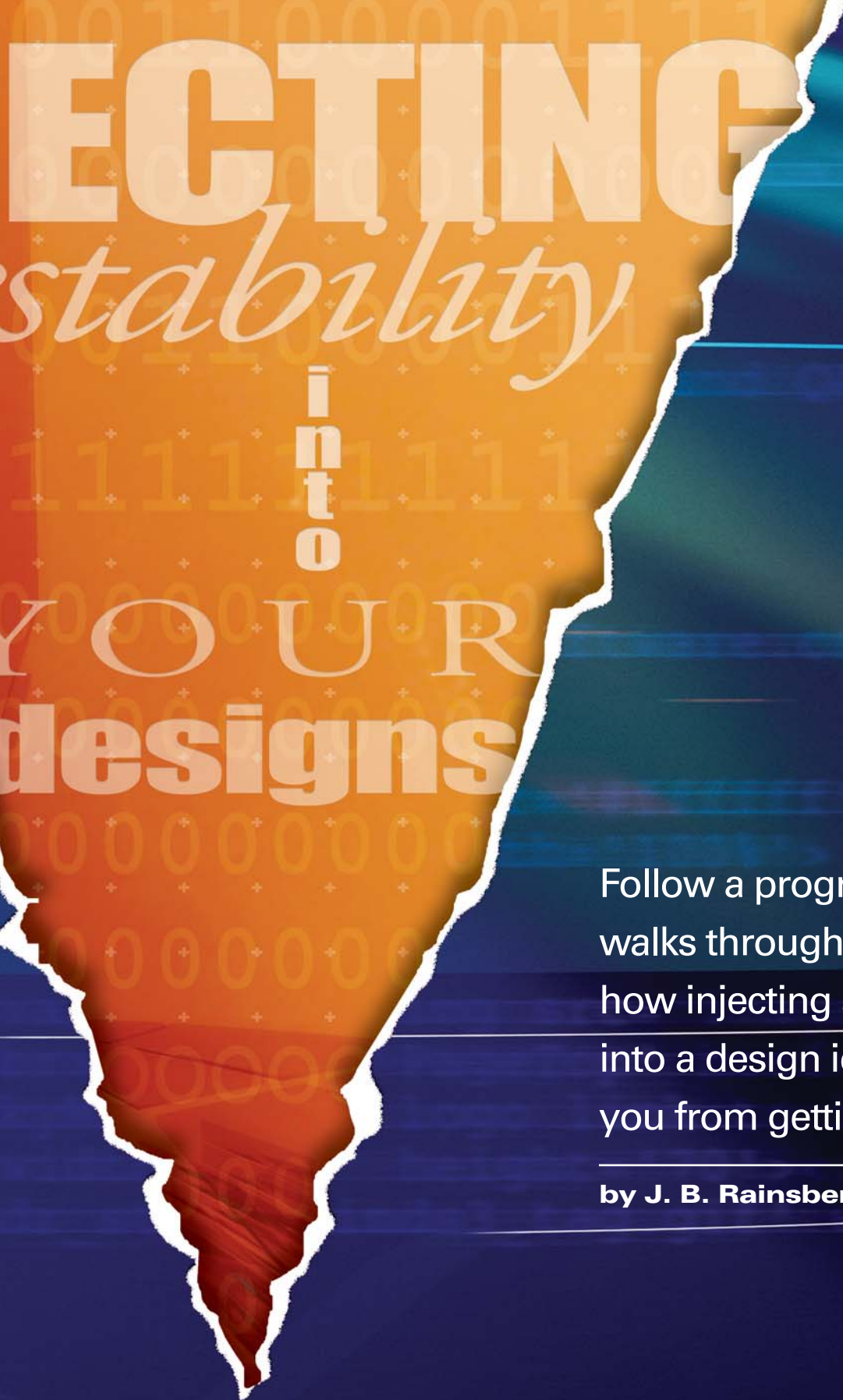


Follow a programmer as he walks through an example of how injecting a dependency into a design idea can keep you from getting stuck.

by J. B. Rainsberger

The term *dependency injection* has attained buzzword status within the programming community. As with any idea that gains so much visibility so quickly, it has its detractors—mainly those who fear the idea will become another golden hammer. Their fears are well founded, as we saw with the book *Design Patterns*. (see Sticky Notes for reference). Shortly after it appeared, programmers began writing systems riddled with global data, claiming to be improving the design with the Singleton pattern. These programmers likely skimmed the pattern description, failed to read about the drawbacks, and then went merrily on their way. One way to avoid repeating this pattern is to see a variety of both good and bad examples of how to apply a design idea. I would like to contribute to this cause by presenting an example of how to improve the testability of a design by injecting a dependency.

Suppose we are building a website to rival PayPal.com by providing a better electronic funds transfer service. Naturally, our site is transferbuddy.biz, taking full advantage of one of the cool new top-level domains. Before we can hope to compete with the big players in this market space, we need to implement a simple transfer of funds between members of our site. I start by test-driving a basic transfer feature, resulting in these tests:

- The Happy Path: The sender has enough funds to cover a transfer to the receiver.
- The sender transfers his entire balance to the receiver, which we allow.
- The sender has insufficient funds to cover the transfer, so it fails.
- The sender attempts to transfer $0, which we disallow.
- The sender attempts to transfer funds to himself, which we disallow.

Each of the tests follows a similar rhythm: create a sender and receiver, execute the transfer, check the receipt, then verify the sender's and receiver's balance. (Amounts are in Canadian dollars only, for now. We'll tackle multiple currencies later.) The receipt indicates either a successful transfer or a denial of the request, including the specific reason. After checking the receipt, the test verifies that the money has (or has not) changed hands. You can find a few of these tests in Listing 1.

As nice as it is to have completed our first feature, transferbuddy.biz is in business to turn a profit, so the next important feature is obvious: We need to collect a fee for each funds transfer. Our crack financial analysis department has identified a fairly simple scheme: Receiving funds is free, but sending funds costs $2 plus 2.9 percent of the value of the transaction. Returning to our Happy Path test, the fee for the successful transfer comes to $4.90 on the $100 transfer. I test-drive this feature, adding a method named `calculateFee()` to the class FundsTransfer. I write three tests, calculating the fee on a $0 transfer, a $1,000 transfer, and amounts such as $157.41 and $157.42 that force me to round down and up, respectively. As I write these tests, I feel a familiar twinge that usually indicates the onset of a design problem. It has to do with the choice of putting the method `calculateFee()` on the class FundsTransfer.

What triggers my vague uneasiness is being forced to deal with irrelevant details when writing these tests. Following is an example:

Test Example

What triggers my vague uneasiness is being forced to deal with irrelevant details when writing these tests. Following is an example:

```
public class CalculateTransferFeeTest extends TestCase {
    public void testNiceRoundFigure() throws Exception {
        Member sender = new Member("jbrains", new BigDecimal("1000.00"));
        Member receiver = new Member("sarah", new BigDecimal("250.00"));

        FundsTransfer fundsTransfer = new FundsTransfer(
                sender, receiver, new BigDecimal("1000.00"));

        BigDecimal actualFee = fundsTransfer.calculateFee();
        assertEquals(new BigDecimal("31.00"), actualFee);
    }
}
```

The test above merely wants to check the fee amount, but that requires creating a `FundsTransfer` object. To do this, I have to supply the sending party, the receiving party, and the amount of the transfer, even though the sender and receiver have no bearing on how the fee is calculated. These irrelevant objects make the test unnecessarily complex. Using them in the other fee calculation tests makes it worse: *duplicate* irrelevant details. The tests are longer, slower, and involve more concepts than they need, creating several kinds of waste: extra time to write the tests, extra time to execute them, extra difficulty in understanding them, and extra maintenance work when things change. So what can I do? I could move `calculateFee()` onto its own production class, but that seems to create unnecessary complexity. We would have two production classes where one ought to suffice. If I make that change, would I merely be trading one set of problems for another? Unsure exactly how to proceed, I start by removing duplication in the test, hiding the sender and receiver behind the method `calculateFee(amount)`. You can see the resulting tests in Listing 2. This is an improvement, but I can't help but feel that I could do better. Once I have more information about how to improve the design of the production code, I'll do it. I know I usually get that information by writing additional tests, so I'll move on.

Now that we can calculate our transaction fee, we need to apply it to the

funds transfer, at least in those cases where the transfer completes successfully. I suppose it would be easy to change the existing tests for our funds transfer to take the transaction fee into account. I start by changing the Happy Path test: The transaction fee is $4.90 on a transfer of $100, so the sender should end up with $895.10, rather than $900. I change the test, watch it fail, then add the code to make it pass. In doing so, the Happy Path test passes, but another one fails—the one where the sender attempts to transfer all his money. The failure message was most unexpected.

```
expected <0.00> but was <-31.00>
```

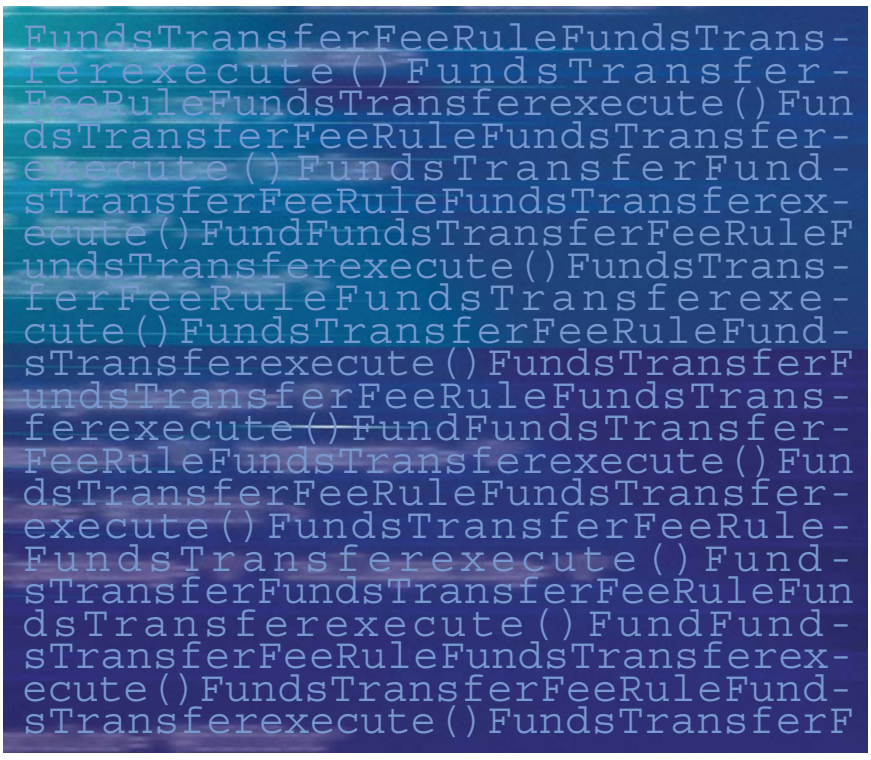
After all my work to avoid the sender's going into the red, it happened anyway. The logic has become more complicated. Now I have to check in advance that the sender has enough money to cover both the transfer and the resulting fee. First, I have to change `testTransferEntireBalance()`, which should expect the transfer now to be denied; then I have to change the code to reflect that fact. After what feels like an eternity (at least three minutes) the comforting green bar returns. In spite of my contentment at seeing the tests pass at 100 percent, I am discouraged by a few observations about the difference between the code now and the code a few minutes ago.

1. Changing the sender's ending balance in the Happy Path test was easy enough, but the changes to the "transfer entire balance" test were sweeping, to say the least.

2. Tomorrow, when our financial analysis department decides on a different transaction fee scheme, I will have to change the sender's ending balance in `testHappyPath()`. Worse, every time the scheme changes, I will have to change both the transfer-processing tests and the fee-calculating tests.

3. Next week, when our financial analysis department cancels the transaction fee scheme, opting instead for a regular membership fee and revenue from banner advertising, I will have to change all the transfer-processing tests back to the way they looked before we ever had transaction fees.

4. One of our useful boundary tests—trying to transfer the entire sender's balance—has effectively disappeared because that test now expects the transfer to fail. I suppose I could replace that test with one that actually transfers the entire balance (leaving enough money to pay the fee), but the exact amount I would need to transfer depends on the transaction fee, so every time the fee rule changes… well, I'd rather not think about it.



This seems to me to be the kind of ripple effect that object-oriented design techniques are meant to help me avoid. Even though the production code design *looks* simple, in the span of a relatively short programming session, the tests have become unnecessarily complex and brittle. In a simple design, dependencies are explicit and obvious, so we can be aware of the impact of changes. The current design hides the fixed dependency between two separate business rules, and hidden dependencies make it difficult to understand the impact of changes. In this case, a change to the transaction fee scheme requires changes in two different kinds of tests: the ones calculating the fee (that makes sense) and the ones transferring the funds (that surprises me). Is there no limit to how much damage a programmer can do in such a short space of time? Apparently not, but there is good news: I have tests, so I can improve the design with confidence. It's time to refactor.

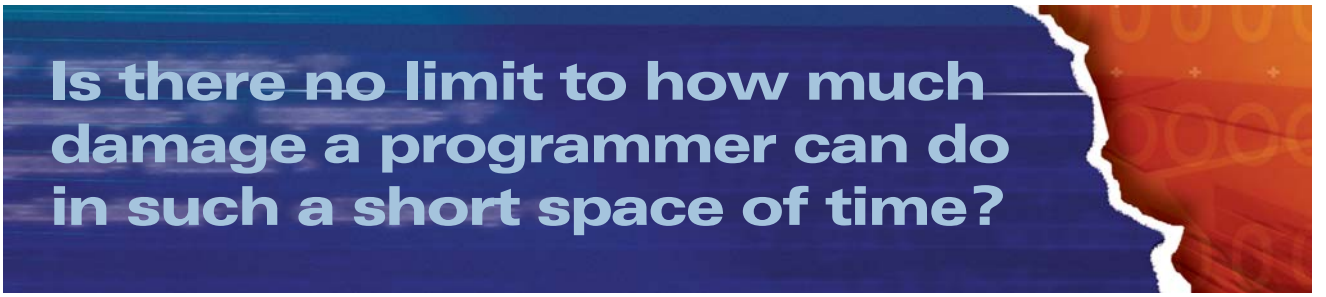
First, I'd like to handle the unnecessary complexity of the tests that calculate the transaction fee. I mentioned hiding some irrelevant details behind a method called calculateFee(amount) in the test class. This new method is considerably easier to use than the existing production code, a conclusion that prompts me to ask myself, *Why isn't the production code that simple?* To fix the problem, I create a new class, FundsTransferFeeRule, and move `calculateFee()` there. (See Listing 3, which shows some "before and after" code.) This simplifies the tests considerably, removing the need to create objects for the sender and receiver of the transfer. It almost makes the tests *too easy*, which I have learned to interpret as a sign that the design is heading in the right direction. This takes care of calculating the fee, but charging the fee still disrupts the tests for performing a simple transfer. This is the point where *dependency injection* enters the scene.

Looking again at the tests, they are brittle. When the fee calculation scheme changes, I have to change the tests related to transferring money rather than just the ones related to calculating the fee. It would be nice if the transfer tests could ignore charging the transaction fee, as that would make the tests more resilient to irrelevant changes in the production code. Imagine if I could change `FundsTransfer` to use any fee-calculating rule I wanted rather than only the implementation provided by `FundsTransferFeeRule`. I could have the basic transfer tests use a fake fee rule that charges $0 per transaction. On the other hand, some tests still need to verify that the fee is properly charged, so they need to use a fee rule that charges *something*, even if it's only a flat rate. How can I have some tests use the $0 fee rule and other tests use, say, a $10 flat rate fee rule? First, I'd need a generic fee rule, so I extract the interface `FeeRule` from `FundsTransferFeeRule`. I can implement this interface differently for each test if I need to. Now I can let the *tests*, and not the `FundsTransfer` object, decide what kind of fee rule to use, by injecting the `FeeRule` into the `FundsTransfer`, rather than allowing it to instantiate the `FeeRule` itself. This is really just a fancy term for providing the FeeRule as a parameter to `FundsTransfer`, turning a hard-coded dependency into a more flexible one. This is the essence of dependency injection: Objects depend on each other without locking themselves into each other.

So what kind of parameter should the fee rule object be? There are three fundamental ways to inject a dependency: *method* injection, *constructor* injection, and *setter* injection. Listing 4 gives you an idea of how the code would change with each of these techniques, discussed by Michael Feathers in his book, *Working Effectively with Legacy Code*. (See Sticky Notes for reference). It might seem strange to apply a legacy code technique to features we are test-driving, but that serves only to drive home an important point about design: A failure to attend to design problems as they emerge equates to writing legacy code for yourself. Our collective experience maintaining ill-designed systems ought to be enough to justify investing time now to avoid serious problems later. If there's one thing you *are* going to need, it's a flexible, testable design.

Of the options in Listing 4, I choose method injection. The method `execute()` is the only part of `FundsTransfer` that needs a `FeeRule`, so it should be the only part that gets one. (See Diagram 1.) I'm following a design principle that recommends keeping the scope of variables as narrow as possible. As the design evolves, if more methods need to use the `FeeRule`, I might promote it from a method parameter to a field, at which point I would have to choose between constructor and setter injection. I'll make that decision when the time comes. With this revised design, I have achieved my goal: The next time I change the fee calculation rule, only the fee-calculating tests need to change. Dependency injection has helped me make that happen.

Still, when I look at the result, `FundsTransfer` has to concern itself with too many things: determining whether the transfer is valid, calculating the transaction fee, determining whether the sender can afford to complete the transaction, then actually moving the money around. Although the design is easier to test, I have that nagging feeling that I'm not finished. That's all right because the design is adequate—maybe even good—for the current needs of the system. Tomorrow, when I'm asked to add more kinds of transactions to the system, I'll have more information to use to improve the design. Until then, I think I'll just call it a day. **{end}**



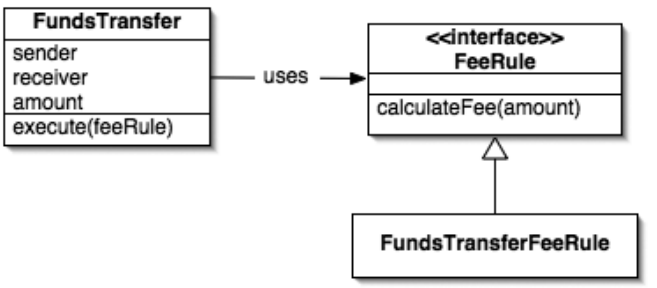


Diagram 1: Injecting the dependency

J. B. (Joe) Rainsberger is the founder of Diaspar Software Services (http://www.diasparsoftware.com), an author, speaker, programmer, teacher, mentor, and host to agile developers from around the world.

Listing 1: Testing an electronic funds transfer

```
[public class SimpleTransferTest extends TestCase {
    private Member sender;
    private Member receiver;

    protected void setUp() throws Exception {
        sender = new Member("jbrains", new BigDecimal("1000.00"));
        receiver = new Member("sarah", new BigDecimal("250.00"));
    }

    public void testHappyPath() throws Exception {
        Receipt receipt = executeFundsTransfer(sender, receiver,
                new BigDecimal("100.00"));

        assertTrue(receipt.showsRequestCompleted());
        assertEquals(new BigDecimal("900.00"), sender.getBalance());
        assertEquals(new BigDecimal("350.00"), receiver.getBalance());
    }

    public void testTransferEntireBalance() throws Exception {
        Receipt receipt = executeFundsTransfer(sender, receiver,
                new BigDecimal("1000.00"));

        assertTrue(receipt.showsRequestCompleted());
        assertEquals(new BigDecimal("0.00"), sender.getBalance());
        assertEquals(new BigDecimal("1250.00"), receiver.getBalance());
    }

    public void testInsufficientFunds() throws Exception {
        Receipt receipt = executeFundsTransfer(sender, receiver,
                new BigDecimal("1000.01"));

        assertTrue(receipt.showsRequestDenied());
       assertEquals(RequestDeniedReceipt.forInsufficientFunds(), receipt);

        assertBalancesUnchanged();
    }

    private Receipt executeFundsTransfer(Member sender, Member receiver,
            BigDecimal amount) {

        FundsTransfer fundsTransfer = new FundsTransfer(sender, receiver,
                amount);
        return fundsTransfer.execute();
    }

    private void assertBalancesUnchanged() {
        assertEquals(new BigDecimal("1000.00"), sender.getBalance());
        assertEquals(new BigDecimal("250.00"), receiver.getBalance());
    }
}
```

Snippet 1: The test knows more than it needs to

```
public class CalculateTransferFeeTest extends TestCase {
    public void testNiceRoundFigure() throws Exception {
        Member sender = new Member("jbrains", new BigDecimal("1000.00"));
        Member receiver = new Member("sarah", new BigDecimal("250.00"));

        FundsTransfer fundsTransfer = new FundsTransfer(
                sender, receiver, new BigDecimal("1000.00"));

        BigDecimal actualFee = fundsTransfer.calculateFee();
        assertEquals(new BigDecimal("31.00"), actualFee);
    }
}
```

Listing 2: Refactoring the tests instead of the production code

```
public class CalculateTransferFeeTest extends TestCase {
    private Member sender;
    private Member receiver;

    protected void setUp() throws Exception {
        sender = new Member("jbrains", new BigDecimal("1000.00"));
        receiver = new Member("sarah", new BigDecimal("250.00"));
    }
    public void testZeroAmount() throws Exception {
        assertEquals(new BigDecimal("2.00"),
                calculateFee(new BigDecimal("0.00")));
    }

    public void testNiceRoundFigure() throws Exception {
        assertEquals(new BigDecimal("31.00"), calculateFee(new BigDecimal(
                "1000.00")));
    }

    public void testRequireRounding() throws Exception {
        assertEquals(new BigDecimal("6.56"), calculateFee(new BigDecimal(
                "157.41")));
        assertEquals(new BigDecimal("6.57"), calculateFee(new BigDecimal(
                "157.42")));
    }

    private BigDecimal calculateFee(BigDecimal amount) {
        FundsTransfer fundsTransfer = new FundsTransfer(sender, receiver,
                amount);
        BigDecimal actualFee = fundsTransfer.calculateFee();
        return actualFee;
    }
}
```

Listing 3: Simpler tests for calculating the transaction fee

Before...

```
public void testNiceRoundFigure() throws Exception {
    Member sender = new Member("jbrains", new BigDecimal("1000.00"));
    Member receiver = new Member("sarah", new BigDecimal("250.00"));

    FundsTransfer fundsTransfer = new FundsTransfer(
            sender, receiver, new BigDecimal("1000.00"));
    BigDecimal transactionFee = fundsTransfer.calculateFee();

    assertEquals(new BigDecimal("31.00"), transactionFee);
}
```

After...

```
public void testNiceRoundFigure() throws Exception {
    FundsTransferFeeRule feeRule = new FundsTransferFeeRule();
      BigDecimal transactionFee = feeRule.calculateFee(new BigDeci-
mal("1000.00"));
    assertEquals(new BigDecimal("31.00"), transactionFee);
}
```

Listing 4: The different injection techniques

Injecting the fee rule through the constructor:

```
FundsTransferFeeRule feeRule = new FundsTransferFeeRule();
FundsTransfer fundsTransfer = new FundsTransfer(feeRule);
fundsTransfer.execute();
```

Injecting the fee rule through a "setter" method:

```
FundsTransferFeeRule feeRule = new FundsTransferFeeRule();
FundsTransfer fundsTransfer = new FundsTransfer();
fundsTransfer.setFeeRule(feeRule);
fundsTransfer.execute();
```

Injecting the fee rule through a method parameter:

```
FundsTransfer fundsTransfer = new FundsTransfer();
FundsTransferFeeRule feeRule = new FundsTransferFeeRule();
fundsTransfer.execute(feeRule);
```