Evolution of Test and Code Via Test-First Design

Jeff Langr

March 2001

Abstract

Test-first design is one of the mandatory practices of Extreme Programming (XP). It requires that programmers do not write any production code until they have first written a unit test. By definition, this technique results in code that is testable, in contrast to the large volume of existing code that cannot be easily tested. This paper demonstrates by example how test coverage and code quality is improved through the use of test-first design.

Approach: An example of code written without the use of automated tests is presented. Next, the suite of tests written for this legacy body of code is shown. Finally, the author iterates through the exercise of completely rebuilding the code, test by test. The contrast between both versions of the production code and the tests is used to demonstrate the improvements generated by virtue of employing test-first design.

Specifics: The code body represents a CSV (comma-separated values) file reader, a common utility useful for reading files in the standard CSV format. The initial code was built in Java over two years ago. Unit tests for this code were written recently, using JUnit (http://www.junit.org) as the testing framework. The CSV reader was subsequently built from scratch, using JUnit as the driver for writing the tests first. The paper presents the initial code and subsequent tests wholesale. The test-first code is presented in an iterative approach, test by test.

Author

Jeff Langr is a consultant with Object Mentor, Inc., responsible for mentoring development teams in XP and training in OO and XP practices. Jeff has over eighteen years software development experience, including close to ten years experience in object-oriented development. Langr is the author of the book Essential Java Style (Prentice Hall, 1999).

Acknowledgments

Bob Koss and Bob Martin, of Object Mentor, provided useful feedback on the paper. Ann Anderson provided pair programming time.

Introduction

In 1998, I was a great Java programmer. I wrote great Java code. Evidence of my great code was the extent to which I thought it was readable and easily maintained by other developers. (Never mind that the proof of this robustness was nonexistent, the distinction of greatness being held purely in *my* head.) I took pride in the great code I wrote, yet I was humble enough to realize that my code might actually break, so I typically wrote a small body of semi-automatic tests subsequent to building the code.

Since 1998, I have been exposed to Extreme Programming (XP). XP is an “agile,” or lightweight, development process designed by Kent Beck. Its chief focus is to allow continual delivery of business value to customers, via software, in the face of uncertain and changing requirements – the reality of most development environments. XP achieves this through a small, minimum set of simple, proven development practices that complement each other to produce a greater whole. The net result of XP is a development team able to produce software at a sustainable and consistently measurable rate.

One of the practices in XP is test-first design (TfD). Adopting TfD means that you write unit-level tests for every piece of functionality that could possibly break. It also means that these tests are written prior to the code. Writing tests before writing code has many effects on the code, some of which will be demonstrated in this paper.

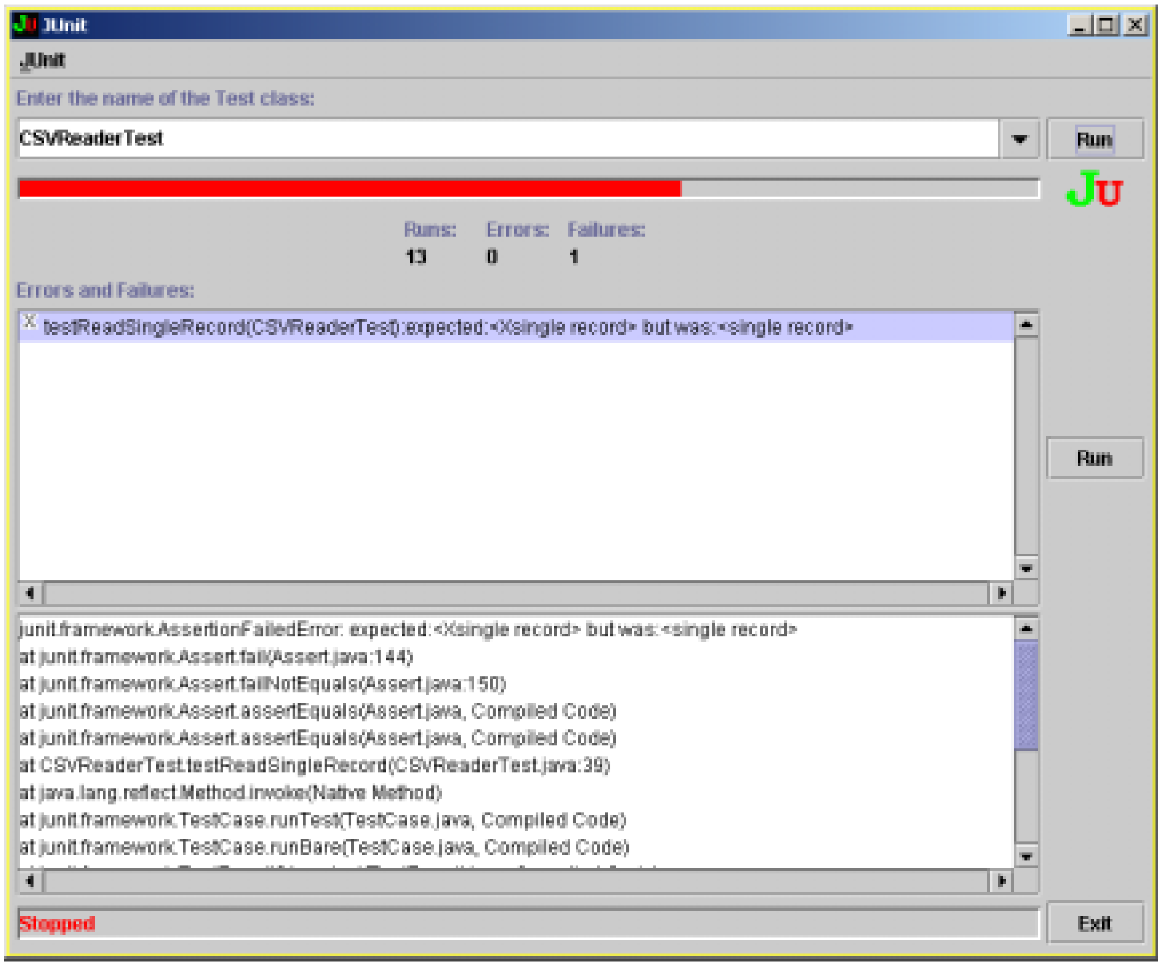
The first (hopefully obvious) effect of TfD, is that the code ends up being testable – you’ve already written the test for it. In contrast, it is often extremely difficult, if not impossible, to write effective unit tests for code that has already been written without consideration for testing. Often, due to the interdependencies of what are typically poorly organized modules, simple unit tests cannot be written without large amounts of context.

Secondly, the process of determining how to test the code can be the more difficult task – once the test is designed, writing the code itself is frequently simple. Third, the granularity of code chunks written by a developer via TfD is much smaller. This occurs because the easiest way to write a unit test is to concentrate on a small discrete piece of functionality. By definition, the number of unit tests thus increases – having smaller code chunks, each with its own unit test, implies more overall code chunks and thus more overall unit tests. Finally, the process of developing code becomes a continual set of small, relatively consistent efforts: write a small test, write a small piece of code to support the test. Repeat.

TfD also employs another important technique that helps drive the direction of tests: tests should be written so that they fail first. Once a test has proven to fail, code is written to make the test pass. The immediate effect of this technique is that testing coverage is increased; this too will be demonstrated in the example section of this paper.

XP’s preferred enabling mechanism for TfD is XUnit, a series of open-source tools available for virtually all OO (and not quite OO) languages and environments: Java, C++, Smalltalk, Python, TCL, Delphi, Perl, Visual Basic, etc. The Java implementation, JUnit, provides a framework on which to build test suites. It is available at http://www.junit.org. A test suite is comprised of many test classes, each of which generally tests a single class of actual production code. A test class contains many discrete test methods, which each establish a test context and then assert actual results against expected results.

JUnit also provides a simple user interface that contains a progress bar showing the success or failure of individual test methods as they are executed. Details on failed tests are shown in other parts of the user interface. Figure 1 presents a sample JUnit execution.



Enter the name of the Test class: CSVReaderTest

Runs: 13 Errors: 0 Failures: 1

Errors and Failures:

```
testReadSingleRecord(CSVReaderTest):expected:<Xsingle record> but was:<single record>
```

```
junit.framework.AssertionFailedError: expected:<Xsingle record> but was:<single record>
at junit.framework.Assert.fail(Assert.java:144)
at junit.framework.Assert.failNotEquals(Assert.java:150)
at junit.framework.Assert.assertEquals(Assert.java, Compiled Code)
at junit.framework.Assert.assertEquals(Assert.java, Compiled Code)
at CSVReaderTest.testReadSingleRecord(CSVReaderTest.java:39)
at java.lang.reflect.Method.invoke(Native Method)
at junit.framework.TestCase.runTest(TestCase.java, Compiled Code)
at junit.framework.TestCase.runBare(TestCase.java, Compiled Code)
```

Stopped

Figure 1 – JUnit user interface

The key part of JUnit is that it is intended to produce Pavlovian responses: a green bar signifies that all tests ran successfully. A red bar indicates at least one failure. Green = good, red = bad. The XP developer quickly develops a routine around deriving a green bar in a reasonably short period of time – perhaps 2 to 10 minutes. The longer it takes to get a green bar, the more likely it is that the new code will introduce a defect. We can usually assume that the granularity of the unit test was too large. Ultimately, the green bar conditioning is to get the developer to learn to build tests for a smaller piece of functionality. Within this paper, references to "getting a green bar" are related to the stimulus-response mechanism that JUnit provides.

Background

During my period of greatness in 1998, I wrote a simple Java utility class, CSVReader, whose function was to provide client applications a simple interface to read and manipulate comma-separated values (CSV) files. I have recently found reason to unearth the utility for potential use in an XP environment.

However, XP doesn't take just anybody's great code. It insists that it come replete with its corresponding body of unit tests. I had no such set of rigorous unit tests. In a vain attempt to satisfy the XP needs, I wrote a set of unit tests against this body of code. The set of tests seemed relatively complete and reasonable. But the code itself, I realized, was less than satisfying.

This revelation came about from attempting to change the functionality of the parsing. Embedded double quotes should only be allowed in a field if they are escaped, i.e. `\"`. The existing functionality allowed embedded double quotes without escaping ("naked" quotes), which leads to some relatively difficult parsing code.

I had chosen to implement the CSVReader using a state machine. The bulk of the code, to parse an individual line, resided in the 100+ line method `columnsFromCSVRecord` (which I had figured on someday refactoring, of course). The attempt to modify functionality was a small disaster: I spent over an hour struggling with the state machine code before abandoning it.

I chose instead to rebuild the CSVReader from scratch, fully using TfD, taking careful note of the small, incremental steps involved. The last section of this paper presents these steps in gory detail, explaining the rationale behind the development of the tests and corresponding code. The next section neatly summarizes the important realizations from the detail section.

Realizations

Building Java code via TfD takes the following sequence:

- Design a test that should fail.
- Immediate failure may be indicated by compilation errors. Usually this is in the form of a class or method that does not yet exists.
- If you had compilation errors, build the code to pass compilation.
- Run all tests in JUnit, expecting a red bar (test failure).
- Build the code needed to pass the test.
- Run all tests in JUnit, expecting a green bar (test success). Correct code as needed until a green bar is actually received.

Building the code needed to pass the test is a matter of building only what is necessary. In many cases, this may involve hard-coding return values from methods. This is a temporary solution. The hard-coding is eliminated by adding another test for additional functionality. This test should break, and thus require a solution that cannot be based on hard-coding.

Design will change. In the CSVReader example, my first approach was to use substring methods to break the line up. This evolved to a StringTokenizer-based solution, then to its current implementation using a state machine. The time required to go from design solution to the next was minimal; I was able to maintain green bars every few minutes. The evolution of tests quickly shaped the ultimate design of the class. The substring solution sufficed for a single test against a record with two columns. But it lasted only minutes, until I designed a new test that introduced records with multiple columns.

The initial attempt to introduce the complexity of the state machine was a personal failure due to my deviation from the rules of TfD. I unsuccessfully wrote code for 20 minutes trying to satisfy a single test. My course correction involved stepping back and thinking about the quickest means of adding a test that would give me a green bar. This involved thinking about a state machine at its most granular level. Given one state and an event, what should the new state be? My test code became repetitions of proving out the state machine at this granularity.

The original code written in 1998 had 6 methods, the longest being well over 100 lines of code. I wrote 15 tests after the fact for this code. I found it difficult to modify functionality in this code. The final code had 23 methods, the longest being 18 source lines of code. I wrote 20 tests as part of building CSVReader via TfD.

Disclaimers

The CSVReader tests are a bit awkward, requiring that a reader be created with a filename, even though the tests are in-memory (specifically the non-public tests). This suggests that CSVReader is not designed well: fixing this would likely mean that CSVReader be modified to take a stream in its constructor (ignoring it if necessary) instead of just a file.

I ended up testing non-interface methods in an effort to reduce the amount of time between green bars. Is testing non-interface methods a code smell? It perhaps suggests that I break out the state machine code into a separate class. My initial thought is that I'm not going to need the separate class at this point. When and if I get to the point where I write some additional code requiring a similar state machine, I will consider introducing a relevant pattern.

Some of the test methods are a bit large – 15 to 20 lines, with more than a couple assertions. My take on test-first design is that each test represents a usable piece of functionality added. I don't have a problem with the larger test methods, then. Commonality should be refactored, however. CSVReaderTest contains a few utility methods that make the individual tests more concise.

Conclusions

Test-first design has a marked effect on both the resulting code and tests written against that code. TfD promotes an approach of very small increments between receiving positive feedback. Using this approach, my experiment demonstrates that the amount of code required to satisfy each additional assertion is small. The time between increments is very brief; on average, I spent 3-4 minutes between receiving green bars with each new assertion introduced. Functionality is continually increasing at a relatively consistent rate.

TfD and incremental refactoring as applied to this example resulted in 33% more tests. It also resulted in a larger number of smaller, more granular methods. Counting simple source lines of code, the average method size in the original source is 25 lines. The average method size in the TfD-produced source is 5 lines. Small method sizes can increase maintainability, communicability, and extensibility of code. Going by average method size in this specific example, then, TfD resulted in considerable improvement of code quality over the original code. Method sized decreased by a factor of 5.

Maintainability of the code was proven by my last pass (Pass Q, below) at building the CSVReader via TfD. The attempt to modify the original body of code to support quote escaping was a failure, representing more than 20 minutes of effort after which time the functionality had not been successfully added. The code built via TfD allowed for this same functionality to be successfully added to the code in 10 minutes, half the time. (Granted, my familiarity with the evolving code base may have added some to the expediency, but I was also very familiar with the original code by virtue of having written several tests for it after the fact.)

TfD alone will not result in improved code quality. Refactoring of code on a frequent basis is required to keep code easily maintainable. Having a body of tests that proves existing functionality means that code refactoring can be performed with impunity.

The final conclusion I drew from this example is that TfD, coupled with good refactoring, can evolve design rapidly. For the CSVReader, I quickly moved from a rudimentary string indexing solution to a state machine, without the need to take what I would consider backward steps. The amount of code replaced at each juncture was minimal, and perhaps even a necessary part of design discovery, allowing development of the application to move consistently forward.

TfD Detailed Example – The CSVReader Class

Origins

I have included listings of the code (CSVReader.java, circa 1998) as initially written, without the benefit of test-first design (Tfd). I have also included the body of tests (CSVReaderTest.java, 23-Feb-2001) written after the fact for the CSVReader code. These listings appear at the end of this paper, due to their length. They are included for comparison purposes. The remainder of the paper presents the evolution of CSVReader via test-first design.

JUnit Test Classes

Building tests for use with JUnit involves creation of separate test classes, typically one for each class to be tested. By convention, the name of each test class is derived by appending the word "Test" to the target class name (i.e. the class to be tested). Thus the test class name for my CSVReader class is CSVReaderTest.

JUnit test classes extend from junit.framework.TestCase. The test class must provide a constructor that takes as its parameter a string representing an arbitrary name for the test case; this is passed to the superclass. The test class must contain at least one test method before JUnit recognizes it as a test class. Test methods must be declared as

```
public void testMethodName()
```

where **`MethodName`** represents the unique name for the test. Test method names should be descriptive and should summarize the functionality proven by the code contained within. The following code shows a skeletal class definition for CSVReaderTest.

```
import junit.framework.*;
public class CSVReaderTest extends TestCase {
   public CSVReaderTest(String name) {
      super(name);
   }
   public void testAbilityToDoSomething() {
      // ... code to set up test...
      assert(conditional);
   }
}
```

Subsequent listings of tests will assume this outline, and will show only the relevant test method itself. Additional code, including refactorings and instance variables, will be displayed as needed.

Getting Started

The initial test written against a class is usually something dealing with object instantiation, or creation of the object. For my CSVReader class, I know that I want to be able to construct it via a filename representing the CSV file to be used as input. The simplest test I can write at this point is to instantiate a CSVReader with a filename string representing a non-existent file, and expect it to throw an exception. `testCreateNoFile()` includes a little bit of context setup: if there is a file with the bogus filename, I delete it so my test works.

```
public void testCreateNoFile() throws IOException {
   String bogusFilename = "bogus.filename";
   File file = new File(bogusFilename);
   if (file.exists())
      file.delete();
   try {
      new CSVReader(bogusFilename);
      fail("expected IO exception on nonexistent CSV file");
   }
   catch (IOException e) {
      pass();
   }
}
void pass() {}
```

I expect test failure if I do not get an IOException. Note my addition of the no-op method `pass()`. I add this method to allow the code to better communicate that a caught IOException indicates test success.

It is important to note that there is no CSVReader.java source file yet. I write the `testCreateNoFile()` method, then compile it. The compilation fails as expected – there is no CSVReader class. I iteratively rectify the situation: I create an empty CSVReader class definition, then recompile CSVReaderTest. The recompile fails: wrong number of arguments in constructor, IOException not thrown in the body of the try statement. Working through compilation errors, I end up with the following code[1]:

```
import java.io.IOException;
public class CSVReader {
   public CSVReader(String filename) throws IOException {
   }
}
```

This code compiles fine. I fire up JUnit and tell it to execute all the tests in CSVReaderTest. JUnit finds one test, `testCreateNoFile()`. (JUnit uses Java reflection capabilities and assumes all methods named with the starting string “test” are to be executed as tests.) As I expect, I see a red bar and the message “expected IO exception on nonexistent CSV file.” My task is to now write the code to fix the failure. It ends up looking like this:

```
import java.io.*;
public class CSVReader {
   public CSVReader(String filename) throws IOException {
      throw new IOException();
   }
}
```

I execute JUnit again, and get a green bar. I have built just enough code, no more, to get all of my tests (just one for now) to pass.

Pass A – Test Against an Empty File

I need CSVReader to be able to recognize valid input files. I want a test that proves CSVReader does not throw an exception if the file exists. I code `testCreateWithEmptyFile()` to build an empty temporary file.

```
public void testCreateWithEmptyFile() throws IOException {
   String filename = "CSVReaderTest.tmp.csv";
   BufferedWriter writer =
      new BufferedWriter(new FileWriter(filename));
   writer.close();
   CSVReader reader = new CSVReader(filename);
   new File(filename).delete();
}
```

This test fails, since the constructor of CSVReader for now is always throwing an IOException. I modify the constructor code:

```
public CSVReader(String filename) throws IOException {
   if (!new File(filename).exists())
      throw new IOException();
}
```

This passes. I want to extend the semantic definition of an empty file, however. I introduce the `hasNext()` method as part of the public interface of CSVReader. A CSVReader opened on an empty file should return true if this method is called. I add an assertion:

```
assert(!reader.hasNext());
```

[1] By now you’ve hopefully noticed that test code appears on the left-hand side of the page, and actual code appears on the right-hand side of the page. This is a nice convention that is used by William Wake at his web site, XP123.com.

after the construction of the CSVReader object, so that the complete test looks like this:

```
public void testCreateWithEmptyFile() throws IOException {
   String filename = "CSVReaderTest.tmp.csv";
   BufferedWriter writer =
      new BufferedWriter(new FileWriter(filename));
   writer.close();
   CSVReader reader = new CSVReader(filename);
   assert(!reader.hasNext());
   new File(filename).delete();
}
```

The compilation fails ("no such method `hasNext()`"). I build an empty method with the signature `public boolean hasNext()`. The question is, what do I return from it? The answer is, a value that will make my test break. Since the test asserts that calling `hasNext()` against the reader will return false, the simplest means of getting the test to fail is to have `hasNext()` return true. I code it; my compile is finally successful.

As I expect, JUnit gives me a red bar upon running the tests. For now, all that is involved in fixing the code is changing the return value of `hasNext()` from true to false – green bar! The resultant code is shown below.

```
import java.io.*;
public class CSVReader {
   public CSVReader(String filename) throws IOException {
      if (!new File(filename).exists())
         throw new IOException();
   }
   public boolean hasNext() {
      return false;
   }
}
```

Note that the test and corresponding code took under five minutes to write. I wrote just enough code to get my unit test to work – nothing more. This is in line with the XP principle that at any given time, there should be no more functionality than what the tests specify. Or as it's better known, "Do The Simplest Thing That Could Possibly Work." Or as it's more concisely known, "DTSTTCPW." Adherence to this principle during TfD, coupled with constantly keeping code clean via refactoring, is what allows me to realize green bars every few minutes. You will see some examples of refactoring in later tests.

Pass B –Read Single Record

The impetus to write more code comes by virtue of writing a test that fails, usually by asserting against new, yet-to-be-coded functionality. This can often be a thought-provoking, difficult task.

One such way of breaking the tests against CSVReader is to create a file with a single record in it, then use the `hasNext()` method to determine if there are available records. This should fail, since we hard-coded `hasNext()` to return false for the last test (Pass A). The new test method is named `testReadSingleRecord()`.

```
public void testReadSingleRecord() throws IOException {
   String filename = "CSVReaderTest.tmp.csv";
   BufferedWriter writer =
         new BufferedWriter(new FileWriter(filename));
   writer.write("single record", 0, 13);
   writer.write("\r\n", 0, 2);
   writer.close();
   CSVReader reader = new CSVReader(filename);
   assert(reader.hasNext());
   reader.next();
   assert(!reader.hasNext());
   new File(filename).delete();
}
```

If I try to fix the code by returning true from `hasNext()`, then `testCreate()` fails. At this point I will have to code some logic to make `testReadSingleRecord()` work, based on working with the actual file created in the test.

The solution has the constructor of CSVReader creating a BufferedReader object against the file represented by the filename parameter. The first line of the reader is immediately read in and stored in an instance variable, `_currentLine`. The `hasNext()` method is altered to return true if `_currentLine` is not null, false otherwise.

Proving the correct operation of the `hasNext()` method does not mean `testReadSingleRecord()` is complete. The semantics implied by the name of the test method are that we should be able to read a single record out of my test file. To complete the test, I should be able to call a method against CSVReader that reads the next record, and then use `hasNext()` to ensure that there are no more records available.

The method name I chose for reading the next record is `next()` – so far, CSVReader corresponds to the java.util.Iterator interface. Compilation of the test breaks since there is not yet a method named `next()` in CSVReader. The method is added with an empty body. This results in JUnit throwing up a red bar for the test. The final line of code is added to the `next()` method:

```
_currentLine = _reader.readLine();
```

This results in the line being read from the file and stored in the instance variable `_currentLine`. Recompiling and re-running the JUnit tests results in a green bar.

```
import java.io.*;
public class CSVReader {
   public CSVReader(String filename) throws IOException {
      if (!new File(filename).exists())
         throw new IOException();
      _reader = new BufferedReader(
                   new java.io.FileReader(filename));
      _currentLine = _reader.readLine();
   }
   public boolean hasNext() {
      return _currentLine != null;
   }
   public void next() throws IOException {
      _currentLine = _reader.readLine();
   }
private BufferedReader _reader;
private String _currentLine;
}
```

Pass C – Refactoring

One of the rules in XP is that there should be no duplicate lines of code. As soon as you recognize the duplication, you should take the time to refactor it. The longer between refactoring intervals, the more difficult it will be to refactor it. Once again, XP is about moving forward consistently through small efforts. Some specific techniques for refactoring code are detailed in Martin Fowler's book, **Refactoring: Improving the Design of Existing Code** (Addison Wesley Longman, Inc., 1999, Reading, Massachusetts). The chief goal of refactoring is to ensure that the current code always has the optimal, simplest design.

Note that there is currently some duplicate code in both CSVReaderTest and CSVReader. Time for some refactoring. In CSVReader, the line of code:

```
_currentLine = _reader.readLine();
```

appears twice, so it is extracted into the new method `readNextLine`:

```
import java.io.*;
public class CSVReader {
   public CSVReader(String filename) throws IOException {
      if (!new File(filename).exists())
         throw new IOException();
      _reader = new BufferedReader(
                  new java.io.FileReader(filename));
      readNextLine();
   }
   public boolean hasNext() {
      return _currentLine != null;
   }
   public void next() throws IOException {
      readNextLine();
   }
   void readNextLine() throws IOException {
      _currentLine = _reader.readLine();
   }
private BufferedReader _reader;
private String _currentLine;
}
```

Within CSVReaderTest, the two lines required to create the BufferedWriter object are refactored to the `setUp()` method. `setUp()` is a method that is executed by the JUnit framework prior to each test method. There is also a corresponding `tearDown()` method that is executed subsequent to the execution of each test method. I modify the `tearDown()` method to include a line of code to delete the temporary CSV file created by the test.

I extract the two lines to close the writer and create a new method `getReaderAndCloseWriter()`. The new test methods, new instance variables, and modified methods are shown in the following listing.

```
public void setUp() throws IOException {
   filename = "CSVReaderTest.tmp.csv";
   writer = new BufferedWriter(new FileWriter(filename));
}
public void tearDown() {
   new File(filename).delete();
}
public void testCreateWithEmptyFile() throws IOException {
   CSVReader reader = getReaderAndCloseWriter();
   assert(!reader.hasNext());
}
public void testReadSingleRecord() throws IOException {
   writer.write("single record", 0, 13);
   writer.write("\r\n", 0, 2);
   CSVReader reader = getReaderAndCloseWriter();
   assert(reader.hasNext());
   reader.next();
   assert(!reader.hasNext());
}
CSVReader getReaderAndCloseWriter() throws IOException {
   writer.close();
   return new CSVReader(filename);
}
private String filename;
private BufferedWriter writer;
```

Pass D – Read Single Record, continued

The test method `testReadSingleRecord` is incomplete. I'm building a CSV reader. I want to ensure that it is able to return the list of columns contained in each record. For a single record with no commas anywhere, I should be able to get back a list that contains one column. The columns should be returned upon the call to `next()`, so my code should look like:

```
List columns = reader.next();
```

The corresponding assertion is:

```
assertEquals(1, columns.size());
```

I insert these two lines in `testReadSingleRecord`:

```
public void testReadSingleRecord() throws IOException {
   writer.write("single record", 0, 13);
   writer.write("\r\n", 0, 2);
   CSVReader reader = getReaderAndCloseWriter();
   assert(reader.hasNext());
   List columns = reader.next();
   assertEquals(1, columns.size());
   assertEquals("single record", columns.get(0));
   assert(!reader.hasNext());
}
```

and compile. The failed compile forces me to modify `next()` to return a java.util.List object. For now, to get the compile to pass, I have `next()` simply return a new ArrayList object. Running JUnit results in a red bar since the size of an empty ArrayList is not 1. I modify `next()` to add an empty string to the ArrayList before it is returned. JUnit now gives me a green bar.

Now I need to ensure that the single column returned from `next()` contains the data I expect ("single record"):

```
assertEquals("single record", columns.get(0));
```

This fails, as expected, so instead of adding an empty string to the return ArrayList, I add the string "single record." I get a green bar. Here's the modified `next()` method:

```
public List next() throws IOException {
   readNextLine();
   List columns = new ArrayList();
   columns.add("single record");
   return columns;
}
```

On the surface, these steps seem unnecessary and even ridiculous. Why am I creating hard-coded solutions? XP promotes the concept that we should build just enough software at any given time to get the job done: DTSTTCPW. The code I have written is just enough to satisfy the tests I have designed. Functionality is added by creating tests to demonstrate that the code does not yet meet that additional desired functionality. Code is then written to provide the missing functionality. The baby steps taken allow for a more consistent rate in delivering additional functionality.

Pass E – Read Two Records

To break `testReadSingleRecord()` I can write two records, each with different data, to the CSV file. While writing `testReadTwoRecords`, I had to recode the nasty pairs of lines required to write each string to the BufferedWriter. I decided to factor that complexity out into the method `writeln`. I subsequently went back and modified the code in `testReadSingleRecord()` to also use the utility method `writeln`.

```
public void testReadTwoRecords() throws IOException {
   writeln("record 1");
   writeln("record 2");
   CSVReader reader = getReaderAndCloseWriter();
   reader.next();
   List columns = reader.next();
   assertEquals("record 2", columns.get(0));
}
// ...
void writeln(String string) throws IOException {
   writer.write(string, 0, string.length());
   writer.write("\r\n", 0, 2);
}
```

In order to fix this broken test scenario, I could go on and keep storing data in the ArrayList, but that would be repeating myself. It's time to write some real code.

To get things to work, the List of columns in the `next()` method is populated with `_currentLine`. Note that the contents of `_currentLine` must be used before they are replaced with the next line; i.e., the columns are populated *before* the call to `readNextLine()`.

```
public List next() throws IOException {
   List columns = new ArrayList();
   columns.add(_currentLine);
   readNextLine();
   return columns;
}
```

Pass F – Two Columns

I'm now at the point where I want to start getting into the CSV part of things. I build `testTwoColumns()`, which tests against a single record with an embedded comma. I expect to get two columns in return, each with the appropriate string data. The test breaks since I am currently assuming that the entire line is a single column.

```
public void testTwoColumns() throws IOException {
   writeln("column 1,column 2");
   CSVReader reader = getReaderAndCloseWriter();
   List columns = reader.next();
   assertEquals(2, columns.size());
   assertEquals("column 1", columns.get(0));
   assertEquals("column 2", columns.get(1));
}
```

To get my green bar, the "simplest thing that could possibly work" is to use the java.lang.String method `substring` to determine the location of any existing comma. I can write that code:

```
public List next() throws IOException {
   List columns = new ArrayList();
   int commaIndex = _currentLine.indexOf(",");
   if (commaIndex == -1)
      columns.add(_currentLine);
   else
   {
      columns.add(_currentLine.substring(0, commaIndex));
      columns.add(_currentLine.substring(commaIndex + 1));
   }
   readNextLine();
   return columns;
}
```

Pass G – Multiple Columns

Breaking a line into two columns is simple enough. A test to see if a line can be split into three or more columns fails.

```
public void testMultipleColumns() throws IOException {
   writeln("column 1,column 2,column 3");
   CSVReader reader = getReaderAndCloseWriter();
   List columns = reader.next();
   assertEquals(3, columns.size());
   assertEquals("column 1", columns.get(0));
   assertEquals("column 2", columns.get(1));
   assertEquals("column 3", columns.get(2));
}
```

Trying to extend the current `substring` solution ends up being too complex. Using a StringTokenizer to split columns on a comma boundary is an easy, elegant solution.

```
public List next() throws IOException {
   List columns = new ArrayList();
   StringTokenizer tokenizer =
      new StringTokenizer(_currentLine, ",");
   while (tokenizer.hasMoreTokens())
      columns.add(tokenizer.nextToken());
   readNextLine();
   return columns;
}
```

I had only spent a few minutes on the substring-based solution, and it worked for the time being, so I don't consider its departure as the mark of a poor initial design decision.

Pass H – State Machine

I write `testCommaInDoubleQuotes()` to allow CSVReader to treat commas as data, not delimiters, if they appeared in a column flanked by double quotes.

```
public void testCommaInDoubleQuotes() throws IOException {
   writeln("\"column with a , (comma)\",column 2");
   CSVReader reader = getReaderAndCloseWriter();
   List columns = reader.next();
   assertEquals(2, columns.size());
   assertEquals("column with a , (comma)", columns.get(0));
   assertEquals("column 2", columns.get(1));
}
```

After thinking for a minute, I realize that the StringTokenizer solution is going to be too difficult to go further with, if even feasible at all. Instead I come up with the idea of a simple state machine, just like in my 1998 solution.

I work on the state machine code after sketching a quick state diagram. It takes about 20 minutes, far longer than I expect, and far too long without any feedback. I make a couple transliteration errors between the table and the code, thus requiring some debugging steps. I decide that my approach – to not build the state code incrementally – is in error. The code I am building to meet the requirements of the state diagram is looking like the old code I wrote. I have one large, ugly method.

At this point I choose to start over again, deleting the state code and trying to see how quickly I can get to a green bar. What this means, though, is that I have to back up and comment out `testCommaInDoubleQuotes()`, adding instead incremental tests that interact with non-interface methods[2].

The simplest state machine to build at this point is one that accepts a single word. This state machine is detailed in Table 1.

Table 1

State	Event	Actions	New State
<init>			delim
delim	any char	append char	inWord
inWord	any char	append char	
inWord	end of string (eos)	writeWord	<fini>

[2] The term non-interface methods is a semantic definition, and indicates methods that are not part of the primary client interface. By converse definition, interface methods are methods that I expect interested clients to interact with – the published behavior per a UML diagram. In Java, if tests reside in the same package as the tested code, the access specifier for these methods is package (default). If the tests reside in a different package than the tested code, the non-interface methods must be designated using the Java keyword `public`.

Technically, non-interface methods become part of the interface, since the test code becomes an interested client.

The easiest way to get going, then, is to track the state of a single word, character by character. I build `testStateOneWord()` to asserts against the initial state of a CSVReader:

```
public void testStateOneWord() throws IOException {
   CSVReader reader = getReaderAndCloseWriter();
   assertEquals(CSVReader.stateDelim, reader.getState());
}
```

To support this in code, I create a new method with package access, `getState()`, and hardcode its return value, `stateDelim`.

```
int getState() {
   return stateDelim;
}
final static int stateDelim = 0;
```

I add a second assertion to build the concept of a current word, which should be empty at this point since I have generated no events:

```
assertEquals("", reader.getCurrentWord());
```

Passing this test involves adding a new method that for now simply returns the empty string.

So how do I track state for given input? Throwing character events at the reader should work. I design the interface into CSVReader to be a method, `charEvent`, that takes a single character as its parameter.

My first assertion, assuming a test word "test", is to throw the single character 't' at the CSVReader object and make sure that my current state is "inWord," per my state table.

```
reader.charEvent('t');
assertEquals(CSVReader.stateInWord, reader.getState());
```

This requires me to add the constant to CSVReader representing the new state. Making the code work means storing the current state as an instance variable (`_state`), initializing it to `stateDelim`, and changing the state to `stateInWord` upon the receiving a `charEvent` message.

```
int getState() {
   return _state;
}
void charEvent(char ch) {
   _state = stateInWord;
}
private int _state = stateDelim;
final static int stateInWord = 1;
```

Next, I loop through the characters in the rest of the test word, sending the appropriate `charEvent` for each. I send the end of string event, which represents a new method. I then assert that my current word is the same as my test word. The complete test method now looks like:

```
public void testStateOneWord() throws IOException {
   CSVReader reader = getReaderAndCloseWriter();
   assertEquals(CSVReader.stateDelim, reader.getState());
   assertEquals("", reader.getCurrentWord());
   reader.charEvent('t');
   assertEquals(CSVReader.stateInWord, reader.getState());
   String testWord = "test";
   for (int i = 1; i < testWord.length(); i++)
      reader.charEvent(testWord.charAt(i));
   reader.endOfStringEvent();
   assertEquals("test", reader.getCurrentWord());
}
```

Fixing this failing test is easy: I add an instance variable `_currentWord`, initialize it to the empty string (“”), and have `endOfStringEvent` set `_currentWord` to the string “test” explicitly. The assertion is that `getCurrentWord()` returns the expected word “test” when all is done. Hardcoding makes it so. The new/modified CSVReader code:

```
String getCurrentWord() {
   return _currentWord;
}
void endOfStringEvent() {
   _currentWord = "test";
}
private String _currentWord = "";
```

Note that I have not even touched the `next()` method – I am testing CSVReader’s ability to maintain my state irrespective of the functionality in `next()`.

Code Smells

To me and others (including reviewers of this paper), testing against non-interface methods is a code smell – a hint that there is something bad about the code. Adding tests against package methods such as `charEvent` and `getCurrentWord()` is such a hint. I am no longer testing against the interface of CSVReader, I am testing against its specific current implementation. This means that the tests will need to be rewritten as the implementation changes.

Ultimately, the smell indicates that the complex state code should be broken into a separate class, perhaps a generic state machine implementation. The new class would have its own set of tests against its public interface. However, my initial reaction is that I’m not going to need the new class for the time being. The effort to split the tests and code out will be roughly the same now or later, so per XP, I will defer the design decision until I really need it.

Pass I – Two Columns

Tracking two columns ended up being the most involved test in the completed application. Building this iteratively took perhaps 20 minutes, but I added my assertions in incrementally, ensuring that I was getting a green bar every few minutes.

The updated state table appears as Table 2.

Table 2

State	Event	Actions	New State
<init>			delim
delim	any char	append char	inWord
inWord	,	writeWord, newWord	delim
inWord	any other char	append char	
inWord	end of string (eos)	writeWord	<fini>

Building `testStateTwoColumns()` assertion by assertion is similar to the technique in Pass H. I create a test input string that should break into two columns. I loop through the string until I receive a comma. I assert that the current column contains the first word in the input string. After sending the comma `charEvent`, I assert that the new state is `stateDelim`. I add a new non-interface method, `getCurrentLineColumns()`, to ensure that each word is added to a list of columns (that will ultimately be returned by the `next()` method). I loop through the rest of the input string, adding assertions as appropriate.

Rather than detail the code evolution here, I have added comments to the test code below. I generally would not leave these comments in the released test. Note that I did a minor refactoring: I decided I didn't care for the term term "word" when I really meant "column." I modified `testStateOneWord()` and `testStateTwoColumns()` accordingly to refer to a `currentColumn` instead of a `currentWord`.

```
public void testStateTwoColumns() throws IOException {
   String testInput = "word1,word2";
   CSVReader reader = getReaderAndCloseWriter();
   int commaIndex = testInput.indexOf(",");
   for (int i = 0; i < commaIndex; i++)
      reader.charEvent(testInput.charAt(i));

   // fixing this means adding a StringBuffer (_currentBuffer)
   // to track the characters as they are appended.
   // getCurrentWord, instead of returning _currentWord,
   // returns _currentBuffer.toString().
   assert(reader.getCurrentColumn().equals("word1"));

   reader.charEvent(',');
   // adding the next assertion means adding in an if
   // statement in charEvent to manage the distinction
   // between stateDelim and stateInWord
   assert(reader.getState() == CSVReader.stateDelim);

   // we also want to make sure the "write word" action
   // takes place. getColumns for now just hardcodes a
   // list with the single entry "word1".
   List columns = reader.getCurrentLineColumns();
   assertEquals(1, columns.size());

   reader.charEvent(testInput.charAt(commaIndex + 1));
   assertEquals(CSVReader.stateInWord, reader.getState());
   // this works.  so we need a test that fails, instead

   // the next assertion requires that _currentBuffer be
   // cleared out, so we add a new method newWord() to
   // blow away the buffer, when comma event is received
   // in inWord state.
   assertEquals("w", reader.getCurrentColumn());

   // the next assertion works.
   for (int i = commaIndex + 2; i < testInput.length(); i++)
      reader.charEvent(testInput.charAt(i));
   reader.endOfStringEvent();
```

```
    assertEquals("word2", reader.getCurrentColumn());

    // do we have our columns?
    columns = reader.getCurrentLineColumns();
    // to get the following to work, we have to make columns
    // into an instance variable. In order to map to the state
    // diagram, we also make a new method writeWord.  writeWord
    // gets the current word and adds it to the columns.  The
    // writeWord method must be called from
    // charEvent->stateInWord->',' and also from endOfStringEvent.
    // At this point, also, the instance variable _currentWord can
    // be removed, along with any references to it.
    // The contents of the columns are also tested, though both
    // tests pass immediately.
    assertEquals(2, columns.size());
    assertEquals("word1", columns.get(0));
    assertEquals("word2", columns.get(1));
}
```

The modified/new CSVReader code resulting from the incremental creation of `testStateTwoColumns()` is shown below.

```
String getCurrentColumn() {
   return _columnBuffer.toString();
}
List getCurrentLineColumns() {
   return _columns;
}
void charEvent(char ch) {
   if (_state == stateInWord) {
      if (ch == ',') {
         writeWord();
         newWord();
         _state = stateDelim;
      }
      else
         append(ch);
   }
   else
      if (_state == stateDelim) {
         _state = stateInWord;
         append(ch);
      }
}
void writeWord() {
   _columns.add(getCurrentColumn());
}
void newWord() {
   _columnBuffer.delete(0, _columnBuffer.length());
}
void append(char ch) {
   _columnBuffer.append(ch);
}
void endOfStringEvent() {
   _currentColumn = "test";
   writeWord();
}
private String _currentColumn = "";
private StringBuffer _columnBuffer = new StringBuffer();
private List _columns = new ArrayList();
```

Pass J – Plugging In The State Machine

Now that I have confidence enough in the state machine to be able to pass the above test situations, I need to hook the state stuff up to the existing framework. But how do I write a test that fails first? The test method `testStateWithRead()`writes a single record and calls the `next()` method against the CSVReader.

```
public void testStateWithRead() throws IOException {
   writeln("record 1,x");
   CSVReader reader = getReaderAndCloseWriter();
   reader.next();
   List columns = reader.getCurrentLineColumns();
   assertEquals(2, columns.size());
}
```

The assertion that fails initially is to test the non-interface `getCurrentLineColumns()` and ensure that it returns a java.util.List containing two columns. This will force us to hook the state machine code into the `next()` method.

Hooking in the state machine code involves writing a for loop in `next()` to send each character of `_currentLine` as character events, followed by the `endOfStringEvent`. At this point, we should recognize the duplicate code involved in parsing through the string, and subsequently delete the string tokenizing technique in a refactoring of sorts. Doing so, however, breaks `testReadTwoRecords`, so we must fix CSVReader by adding code to clear the columns array and create a new word.

```
public List next() throws IOException {
   _columns.clear();
   newWord();
   for (int i = 0; i < _currentLine.length(); i++)
      charEvent(_currentLine.charAt(i));
   endOfStringEvent();
   readNextLine();
   return getCurrentLineColumns();
}
```

Pass K – Comma In Double Quotes

I retry my `testCommaInDoubleQuotes()` (Pass H), but still fails. Instead of trying to fix it wholesale, I approach the solution incrementally. I first update the state machine table to represent the double-quote functionality I am trying to add (Table 3).

Table 3

State	Event	Actions	New State
<init>			delim
delim	“		inQuoteWord
delim	any other char	append char	inWord

State	Event	Actions	New State
inWord	,	writeWord, newWord	delim
inWord	any other char	append char	
inWord	end of string (eos)	writeWord	<fini>
inQuoteWord	“	writeWord, newWord	quoteInQuoteWord
inQuoteWord	any other char (including ,)	append char	
quoteInQuoteWord	,		delim
quoteInQuoteWord	eos		<fini>

I comment `testCommaInDoubleQuotes()` once more, and create a new test against state, `testDoubleQuotes()`. Again, this test is built iteratively, assert by assert.

```
public void testDoubleQuotes() throws IOException {
   // example: "A,a",b
   CSVReader reader = getReaderAndCloseWriter();
   reader.charEvent('"');
   assertEquals(CSVReader.stateInQuoteWord, reader.getState());
   reader.charEvent('A');
   reader.charEvent(',');
   assertEquals(CSVReader.stateInQuoteWord, reader.getState());
   reader.charEvent('a');
   reader.charEvent('"');
   assertEquals(
      CSVReader.stateQuoteInQuoteWord, reader.getState());
   assertEquals("A,a", reader.getCurrentLineColumns().get(0));
   reader.charEvent(',');
   assertEquals(CSVReader.stateDelim, reader.getState());
   reader.charEvent('b');
   reader.endOfStringEvent();
   assertEquals("b", reader.getCurrentLineColumns().get(1));
}
```

While building `testDoubleQuotes()`, I note that the code in `charEvent` is beginning to get confusing. I choose to refactor now, before it’s too late to do so easily. I reorganize things based on character events. Each special character is treated now as a separate method.

```
void charEvent(char ch) {
   switch (ch) {
      case ',': commaEvent(); break;
      case '"': doubleQuoteEvent(); break;
      default :
         defaultCharEvent(ch); break;
   }
}
void commaEvent() {
   switch (_state) {
      case (stateInWord):
         writeWord();
         newWord();
         _state = stateDelim;
         break;
      case (stateDelim):
         _state = stateInWord;
         append(',');
         break;
      case (stateInQuoteWord):
         append(',');
         break;
      case (stateQuoteInQuoteWord):
         _state = stateDelim;
         break;
   }
}
void doubleQuoteEvent() {
   switch (_state) {
      case stateDelim:
         _state = stateInQuoteWord;
         break;
      case stateInWord:
         append('"');
         break;
      case stateInQuoteWord:
         _state = stateQuoteInQuoteWord;
         writeWord();
         newWord();
         break;
   }
}
void defaultCharEvent(char ch) {
   switch (_state) {
      case stateDelim:
         _state = stateInWord;
         append(ch);
         break;
      case stateInWord:
         append(ch);
         break;
      case stateInQuoteWord:
         append(ch);
         break;
   }
}
final static int stateInQuoteWord = 2;
final static int stateQuoteInQuoteWord = 3;
```

Once I get `testDoubleQuotes()` working, I uncomment `testCommaInDoubleQuotes()` and discover that it now works! I could choose to eliminate one of the two tests at this point, but I feel that the different approaches taken to testing the same functionality – one more “black box” than “white box” – provides better test coverage.

Pass L – More Functionality

Now that the tough part is done, I choose to add some functionality. Comment lines need to be supported. The new test method `testComment()` drives the addition of new functionality:

```
public void testComment() throws IOException {
   writeln("line 1");
   writeln("# comment line");
   writeln("line 2");
   CSVReader reader = getReaderAndCloseWriter();
   reader.next();
   List columns = reader.next();
   assertEquals("line 2", columns.get(0));
}
```

The corresponding code modifications:

```
void readNextLine() throws IOException {
   do
      _currentLine = _reader.readLine();
   while (_currentLine != null && isCommentLine(_currentLine));
}
boolean isCommentLine(String line) {
   return line.charAt(0) == '#';
}
```

While writing `testComment()`, I mentally note the possibility for an IndexOutOfBounds exception to be thrown if a line read is empty. Rather than write the code that I think might be necessary, I decide to write a test (`testEmptyLine()`) to determine if this is indeed the case.

```
public void testEmptyLine() throws IOException
{
   try {
      writeln("");
      CSVReader reader = getReaderAndCloseWriter();
      reader.next();
      pass();
   }
   catch (Exception e) {
      fail(e.getMessage());
   }
}
```

Sure enough, the code throws an exception. I modify the new `isCommentLine()` method in CSVReader to handle the special case:

```
boolean isCommentLine(String line) {
   if (line.length() == 0)
      return false;
   return line.charAt(0) == '#';
}
```

Pass M – Still More Ideas

What if the entire file consists only of comments? Do we get an error? I add `testOnlyComments()`, which passes.

```
public void testOnlyComments() throws IOException {
   writeln("# ...");
   CSVReader reader = getReaderAndCloseWriter();
   assert(!reader.hasNext());
}
```

Not a great idea for a test, since it passed, but I coded it, and I don't have a heartache about leaving it in.

I'm pretty sure that reading too many records via the `next()` method will cause an exception, but probably not the one I want. I code `testEOF()` to call `reader.next()` twice, expecting that the second call generates an IOException.

```
public void testEOF() throws IOException {
   writeln("x");
   CSVReader reader = getReaderAndCloseWriter();
   reader.next();
   try    {
      reader.next();
      fail("should have gotten exception");
   }
   catch (IOException e) {
      pass();
   }
}
```

It doesn't; instead, a NullPointerException is thrown. I modify `next()` to throw an IOException if the current line is null. After that works, I do a minor refactoring since my `next()` method is getting large and doing far too many different things. The new `parse(String)` method comes about via an Extract Method refactoring against `next()`.

```
public List next() throws IOException {
   if (_currentLine == null)
      throw new IOException("Read past end of file in next()");
   parse(_currentLine);
   readNextLine();
   return getCurrentLineColumns();
}
void parse(String line) {
   _columns.clear();
   newWord();
   for (int i = 0; i < line.length(); i++)
      charEvent(line.charAt(i));
   endOfStringEvent();
}
```

Pass N – Whitespace

My current CSVReader implementation does not account for leading and trailing spaces accordingly. Spaces and tabs should be trimmed from both ends of a column, unless the spaces appear within double-quote delimited columns. This requires a modified state diagram (Table 4). The space events are now handled, and `writeWord` is replaced with the action `writeTrimWord` in some circumstances.

Table 4

State	Event	Actions	New State
<init>			delim
delim	space, \t		
delim	"		inQuoteWord
delim	any other char	append char	inWord
inWord	comma	writeTrimWord,	delim

		newWord	
inWord	any other char	append char	
inWord	end of string (eos)	writeTrimWord	<fini>
inQuoteWord	“	writeWord, newWord	quoteInQuoteWord
inQuoteWord	any other char (including comma, space, \t)	append char	
quoteInQuoteWord	comma		delim
quoteInQuoteWord	space, \t		
quoteInQuoteWord	eos		<fini>

I create `testStateWhitespace()` to validate this table.

```
public void testStateWhitespace() throws IOException {
   String testInput = " a ,\tb\t";
   CSVReader reader = getReaderAndCloseWriter();

   reader.charEvent(' ');
   assertEquals(CSVReader.stateDelim, reader.getState());

   reader.charEvent('a');
   reader.charEvent(' ');
   reader.charEvent(',');
   List columns = reader.getCurrentLineColumns();
   assertEquals("a", columns.get(0));

   reader.charEvent('\t');
   reader.charEvent('b');
   reader.charEvent('\t');
   reader.endOfStringEvent();
   columns = reader.getCurrentLineColumns();
   assertEquals("b", columns.get(1));
}
```

The resulting code:

```
void charEvent(char ch) {
   switch (ch) {
      case ',':   commaEvent(); break;
      case '"':   doubleQuoteEvent(); break;
      case ' ':
      case '\t':  whitespaceEvent(ch); break;
      default:    defaultCharEvent(ch); break;
   }
}
void whitespaceEvent(char ch) {
   if (_state == stateDelim)
      ;
   else
      defaultCharEvent(ch);
}
void commaEvent() {
   switch (_state) {
      case (stateInWord):
         writeEndTrimWord();
         newWord();
         _state = stateDelim;
         break;
      case (stateDelim):
         _state = stateInWord;
         append(',');
         break;
      case (stateInQuoteWord):
         append(',');
         break;
      case (stateQuoteInQuoteWord):
         _state = stateDelim;
         break;
   }
}
void endOfStringEvent() {
   if (_state == stateInWord)
      writeEndTrimWord();
   else
      writeWord();
}
void writeEndTrimWord() {
   _columns.add(endTrim(getCurrentColumn()));
}
String endTrim(String source) {
   int i = source.length() - 1;
   while (i > -1)
      if (isWhitespace(source.charAt(i)))
         i--;
      else
         break;
   return source.substring(0, i + 1);
}
boolean isWhitespace(char ch) {
   return ch == ' ' || ch == '\t';
}
```

Pass O – Double Quotes

I write the method `testStateQuotes()`, to ensure that whitespace and quotes were handled properly. This is a bad deviation from plan, evidenced by the fact that all asserts in this test pass on its first execution. In retrospect, I realize that this functionality was added back in Pass N by virtue of coding the state table.

```
public void testStateQuotes() throws IOException {
   //   example string = " \" x, \" , \"y\" ";
   CSVReader reader = getReaderAndCloseWriter();
   sendCharEvents(reader, " \" x, \" ");
   assertEquals(
      CSVReader.stateQuoteInQuoteWord, reader.getState());
   reader.charEvent(',');
   List columns = reader.getCurrentLineColumns();
   assertEquals(" x, ", columns.get(0));
   sendCharEvents(reader, " \"y\" ");
   reader.endOfStringEvent();
   columns = reader.getCurrentLineColumns();
   assertEquals("y", columns.get(1));
}
void sendCharEvents(CSVReader reader, String string) {
   for (int i = 0; i < string.length(); i++)
      reader.charEvent(string.charAt(i));
}
```

The more useful thing that comes out of writing this test is that I finally realize I am writing duplicate test code. Many of the existing test methods call the `charEvent` method over and over. I write the method `sendCharEvents` to take a CSVReader and a String as parameters.

Pass P – Empty Columns

I add `testEmptyFields()` to determine if empty columns are handled correctly.

```
public void testEmptyFields() throws IOException {
   writeln("");
   writeln(",");
   writeln(",a,,,");
   CSVReader reader = getReaderAndCloseWriter();
   List columns = reader.next();
   assertEquals(1, columns.size());
   assertEquals("", columns.get(0));
   columns = reader.next();
   assertEquals(2, columns.size());
   columns = reader.next();
   assertEquals(5, columns.size());
}
```

They are not. I modify the code in `commaEvent()` accordingly.

```
void commaEvent() {
   switch (_state) {
      case (stateInWord):
         writeEndTrimWord();
         newWord();
         _state = stateDelim;
         break;
      case (stateDelim):
         writeWord();
         break;
      case (stateInQuoteWord):
         append(',');
         break;
      case (stateQuoteInQuoteWord):
         _state = stateDelim;
         break;
   }
}
```

When a comma event was received while in `stateDelim`, I was previously changing the state to `stateInWord` and appending the comma. The code now just calls the `writeWord()` method.

I also add `testUnmatchedDoubleQuoteIsAnError()`. If the last column in a record starts with a double quote, but another quote is not received before the end of the string, an error should be thrown.

```
public void testUnmatchedDoubleQuoteIsAnError()
   throws IOException {
   writeln("\"jkl");
   CSVReader reader = getReaderAndCloseWriter();
   try {
      reader.next();
      fail("should have thrown IO exception");
   }
   catch (IOException e) {
      pass();
   }
}
```

The `endOfStringEvent()` method is modified accordingly. It now throws an IOException, and therefore so must the `parse` method.

```
void parse(String line) throws IOException {
   _columns.clear();
   newWord();
   for (int i = 0; i < line.length(); i++)
      charEvent(line.charAt(i));
   endOfStringEvent();
}
void endOfStringEvent() throws IOException {
   switch (_state) {
      case stateInWord:
         writeEndTrimWord();
         break;
      case stateInQuoteWord:
         throw new IOException(
         "Badly formed record: quoted string not terminated");
      default:
         writeWord();
         break;
   }
}
```

Pass Q – Embedded Double Quotes

I wrote the initial CSVReader, allowing for quotes to be embedded within a column, just as long as the string representing a column was flanked with double quotes. This ends up being a poor definition, if not ill-defined in some situations.

My final test is to introduce completely new functionality: I want to allow double-quotes within a column if they are escaped (i.e. prefixed with a backslash), a much simpler implementation.

The state machine has to be modified slightly. Table 5 shows the relevant state modifications only.

Table 5

State	Event	Actions	New State
inQuoteWord	\		escapeInQuote
escapeInQuote	“	append “	inQuoteWord
escapeInQuote	any other char	append \ append char	inQuoteWord

I write `testEmbeddedQuotes()` to build the new state behavior.

```
public void testEmbeddedQuotes() throws IOException {
   // example: " \" x \\\"y \",z"  --> " x \"y"  ,  z
   CSVReader reader = getReaderAndCloseWriter();
   sendCharEvents(reader, " " + quote + " x " + backslash);
   assertEquals(CSVReader.stateEscapeInQuote, reader.getState());
   reader.charEvent(quote);
   assertEquals(CSVReader.stateInQuoteWord, reader.getState());
   sendCharEvents(reader, "y " + quote + ",");
   List columns = reader.getCurrentLineColumns();
   assertEquals(" x \"y ", columns.get(0));
}
final char backslash = '\\';
final char quote = '\"';
```

I also write `testEscapeCharNoQuote()` to ensure that the backslash character is handled correctly if not followed by a double-quote character.

```
public void testEscapeCharNoQuote() throws IOException {
   // example: "\"a\b\"" --> "a\\b"
   CSVReader reader = getReaderAndCloseWriter();
   sendCharEvents(reader, quote + "a" + backslash + "b" + quote);
   assertEquals(
      CSVReader.stateQuoteInQuoteWord, reader.getState());
   reader.endOfStringEvent();
   List columns = reader.getCurrentLineColumns();
   assertEquals("a" + backslash + "b", columns.get(0));
}
```

The code modifications to support these two tests are as follows. Time elapsed to add the last two tests and corresponding code was about ten minutes.

```
void charEvent(char ch) {
   switch (ch) {
      case ',':   commaEvent(); break;
      case '"':   doubleQuoteEvent(); break;
      case ' ':
      case '\t':  whitespaceEvent(ch); break;
      case '\\':   backslashEvent(ch); break;
      default:    defaultCharEvent(ch); break;
   }
}
void backslashEvent(char ch) {
   if (_state == stateInQuoteWord)
      _state = stateEscapeInQuote;
   else
      defaultCharEvent(ch);
}
```

```
void doubleQuoteEvent() {
   switch (_state) {
      case stateDelim:
         _state = stateInQuoteWord;
         break;
      case stateInWord:
         append('"');
         break;
      case stateInQuoteWord:
         _state = stateQuoteInQuoteWord;
         writeWord();
         newWord();
         break;
      case stateEscapeInQuote:
         _state = stateInQuoteWord;
         append('"');
         break;
   }
}
void defaultCharEvent(char ch) {
   switch (_state) {
      case stateDelim:
         _state = stateInWord;
         append(ch);
         break;
      case stateInWord:
         append(ch);
         break;
      case stateInQuoteWord:
         append(ch);
         break;
      case stateEscapeInQuote:
         append('\\');
         append(ch);
         _state = stateInQuoteWord;
         break;
   }
}
final static int stateEscapeInQuote = 4;
```

Final Code

I chose not to include the final set of tests, for space considerations, since they are well represented in the document above. If you are interested in a copy of CSVReaderTest.java, please email me at JLangr@ObjectMentor.com.

The final CSVReader.java code is included, however.

CSVReader.java, built via TfD

```
import java.io.*;
import java.util.*;
public class CSVReader
{
   public CSVReader(String filename) throws IOException {
      if (!new File(filename).exists())
         throw new IOException();
      _reader = new BufferedReader(
                    new java.io.FileReader(filename));
      readNextLine();
   }
   public boolean hasNext() {
      return _currentLine != null;
   }
   public List next() throws IOException {
      if (_currentLine == null)
         throw new IOException(
            "Read past end of file in next()");
      parse(_currentLine);
      readNextLine();
      return getCurrentLineColumns();
   }
   void parse(String line) throws IOException {
      _columns.clear();
      newWord();
      for (int i = 0; i < line.length(); i++)
         charEvent(line.charAt(i));
      endOfStringEvent();
   }
   void readNextLine() throws IOException {
      do
         _currentLine = _reader.readLine();
      while (_currentLine != null &&
             isCommentLine(_currentLine));
   }
   int getState() {
      return _state;
   }
   String getCurrentWord() {
      return _currentWord;
   }
   String getCurrentColumn() {
      return _columnBuffer.toString();
   }
   List getCurrentLineColumns() {
      return _columns;
   }
   void charEvent(char ch) {
      switch (ch) {
         case ',':   commaEvent(); break;
         case '"':   doubleQuoteEvent(); break;
         case ' ':
         case '\t':  whitespaceEvent(ch); break;
         case '\\':   backslashEvent(ch); break;
         default:    defaultCharEvent(ch); break;
      }
   }
   void backslashEvent(char ch) {
      if (_state == stateInQuoteWord)
```

```
         _state = stateEscapeInQuote;
      else
         defaultCharEvent(ch);
   }
   void whitespaceEvent(char ch) {
      if (_state == stateDelim)
         ;
      else
         defaultCharEvent(ch);
   }
   void commaEvent() {
      switch (_state) {
         case (stateInWord):
            writeEndTrimWord();
            newWord();
            _state = stateDelim;
            break;
         case (stateDelim):
            writeWord();
            break;
         case (stateInQuoteWord):
            append(',');
            break;
         case (stateQuoteInQuoteWord):
            _state = stateDelim;
            break;
      }
   }
   void doubleQuoteEvent() {
      switch (_state) {
         case stateDelim:
            _state = stateInQuoteWord;
            break;
         case stateInWord:
            append('"');
            break;
         case stateInQuoteWord:
            _state = stateQuoteInQuoteWord;
            writeWord();
            newWord();
            break;
         case stateEscapeInQuote:
            _state = stateInQuoteWord;
            append('"');
            break;
      }
   }
   void defaultCharEvent(char ch) {
      switch (_state) {
         case stateDelim:
            _state = stateInWord;
            append(ch);
            break;
         case stateInWord:
            append(ch);
            break;
         case stateInQuoteWord:
            append(ch);
            break;
         case stateEscapeInQuote:
            append('\\');
            append(ch);
            _state = stateInQuoteWord;
            break;
      }
   }
   void writeWord() {
      _columns.add(getCurrentColumn());
   }
   void newWord() {
      _columnBuffer.delete(0, _columnBuffer.length());
   }
   void append(char ch) {
      _columnBuffer.append(ch);
   }
   void endOfStringEvent() throws IOException {
      switch (_state) {
         case stateInWord:
            writeEndTrimWord();
            break;
         case stateInQuoteWord:
            throw new IOException(
            "Badly formed record: quoted string not terminated");
         default:
            writeWord();
            break;
      }
   }
   void writeEndTrimWord() {
      _columns.add(endTrim(getCurrentColumn()));
   }
   String endTrim(String source) {
      int i = source.length() - 1;
      while (i > -1)
         if (isWhitespace(source.charAt(i)))
            i--;
         else
            break;
      return source.substring(0, i + 1);
   }
   boolean isWhitespace(char ch) {
      return ch == ' ' || ch == '\t';
   }
   boolean isCommentLine(String line) {
      if (line.length() == 0)
         return false;
      return line.charAt(0) == '#';
   }
```

```
private String _currentColumn = "";
private StringBuffer _columnBuffer = new StringBuffer();
private List _columns = new ArrayList();
private String _currentWord = "";
private BufferedReader _reader;
private String _currentLine;
private int _state = stateDelim;
final static int stateDelim = 0;
final static int stateInWord = 1;
final static int stateInQuoteWord = 2;
final static int stateQuoteInQuoteWord = 3;
final static int stateEscapeInQuote = 4;
}
```

CSVReader.java, circa 1998

```
import java.io.BufferedReader;
import java.io.IOException;
import java.util.List;
public class CSVReader {
   CSVReader(String filename) throws IOException {
      _reader = new BufferedReader(new java.io.FileReader(filename));
      loadNextNonCommentLine();
   }
   public List next() throws IOException {
      if (_line == null)
         throw new IOException("Read past end of file");
      List columns = columnsFromCSVRecord(_line);
      loadNextNonCommentLine();
      return columns;
   }
   public boolean hasNext() {
      return _line != null;
   }
   void loadNextNonCommentLine() throws IOException {
      do
         _line = _reader.readLine();
      while (_line != null && _line.startsWith(COMMENT_SYMBOL));
      if (_line == null)
         _reader.close();
   }
   List columnsFromCSVRecord(String line) throws IOException {
      char state = stateINIT;
      char ch;
      int i = 0;
      List tokens = new java.util.ArrayList();
      StringBuffer buffer = new StringBuffer();
      char[] charArray = line.toCharArray();
      while (i < charArray.length) {
         ch = charArray[i++];
         switch (state) {
            case stateINIT:
               switch (ch) {
                  case '"':
                     buffer.append(ch);
                     state = stateQUOTED_DATA;
                     break;
                  case ',':
                     state = stateNEW_TOKEN;
                     tokens.add(clean(buffer));
                     buffer = new StringBuffer();
                     break;
                  case '\t': case ' ':
                     break;
                  case '#':
                     state = stateCOMMENT;
                     break;
                  default:
                     state = stateDATA;
                     buffer.append(ch);
                     break;
               }
               break;
            case stateCOMMENT:
               break;
            case stateQUOTED_DATA:
               switch (ch) {
                  case '"':
                     buffer.append(ch);
                     state = stateQUOTE_IN_QUOTED_DATA;
                     break;
                  default:
                     buffer.append(ch);
                     break;
               }
               break;
            case stateQUOTE_IN_QUOTED_DATA:
               switch (ch) {
                  case '"':
                     state = stateQUOTED_DATA;
                     break;
                  case ',':
                     state = stateNEW_TOKEN;
                     tokens.add(clean(buffer));
                     buffer = new StringBuffer();
                     break;
                  case ' ': case '\t':
                     break;
                  case '#':
                     tokens.add(clean(buffer));
                     state = stateCOMMENT;
                     break;
                  default:
                     throw new IOException(
                        "badly formed CSV record:" + line);
               }
               break;
            case stateDATA:
               switch (ch) {
                  case '#':
                     tokens.add(clean(buffer));
                     state = stateCOMMENT;
                     break;
                  case ',':
                     state = stateNEW_TOKEN;
                     tokens.add(clean(buffer));
                     buffer = new StringBuffer();
                     break;
                  default:
                     buffer.append(ch);
                     break;
               }
               break;
            case stateNEW_TOKEN:
               switch (ch) {
                  case '#':
                     tokens.add(clean(buffer));
                     state = stateCOMMENT;
                     break;
                  case ',':
                     tokens.add(clean(buffer));
                     buffer = new StringBuffer();
                     break;
                  case ' ': case '\t':
                     state = stateWHITESPACE;
                     break;
                  case '"':
                     buffer.append(ch);
                     state = stateQUOTED_DATA;
                     break;
                  default:
                     state = stateDATA;
                     buffer.append(ch);
                     break;
               }
               break;
            case stateWHITESPACE:
               switch (ch) {
                  case '#':
                     state = stateCOMMENT;
                     break;
                  case ',':
                     state = stateNEW_TOKEN;
                     // ACCEPT NEW EMPTY COLUMN HERE??
                     break;
                  case '"':
                     buffer.append(ch);
                     state = stateQUOTED_DATA;
                     break;
                  case ' ': case '\t':
                     break;
                  default:
                     state = stateDATA;
                     buffer.append(ch);
                     break;
               }
               break;
            default:
               break;
         }
      }
      if (state == stateQUOTED_DATA)
         throw new IOException("Unmatched quotes in line:\n" + line);
      if (state != stateCOMMENT)
         tokens.add(clean(buffer));
      return tokens;
   }
   String clean(StringBuffer buffer) {
      String string = buffer.toString().trim();
      if (string.startsWith(DOUBLE_QUOTE))
         return string.substring(1, string.length() - 1);
      return string;
   }
   private BufferedReader _reader;
   private String _line;
   private static final String DOUBLE_QUOTE = "\"";
   private static final String COMMENT_SYMBOL = "#";
   private static final char stateINIT = 'S';
   private static final char stateCOMMENT = '#';
   private static final char stateQUOTED_DATA = 'q';
   private static final char stateQUOTE_IN_QUOTED_DATA = 'Q';
   private static final char stateDATA = 'D';
   private static final char stateNEW_TOKEN = 'N';
   private static final char stateWHITESPACE = 'W';
}
```

CSVReaderTest.java, 23-Feb-2001

```
import junit.framework.*;
import java.io.*;
import java.util.*;
public class CSVReaderTest extends TestCase
{
   public CSVReaderTest(String name) {
      super(name);
   }
   public void setUp() throws IOException {
      filename = "CSVReaderTest.tmp.csv";
      writer = new BufferedWriter(new FileWriter(filename));
   }
   public void tearDown() throws IOException {
      new File(filename).delete();
   }
   public void testCreate() throws IOException {
      CSVReader reader = getReaderAndCloseWriter();
      assert(!reader.hasNext());
   }
   public void testSingleRecordSingleField() throws IOException {
      writeln("test");
      CSVReader reader = getReaderAndCloseWriter();
      assert(reader.hasNext());
      List columns  = reader.next();
      assertEquals(1, columns.size());
      assertEquals("test", columns.get(0));
   }
   public void testSingleFieldMultipleColumns() throws IOException {
      writeln("test2col1, test2col2");
      CSVReader reader = getReaderAndCloseWriter();;
      assert(reader.hasNext());
      List columns = reader.next();
      assertEquals(2, columns.size());
      assertEquals("test2col1", columns.get(0));
      assertEquals("test2col2", columns.get(1));
   }
   public void testEOF() throws IOException {
      writeln("line1");
      CSVReader reader = getReaderAndCloseWriter();
      reader.next();
      try {
         reader.next();
         fail("expected exception here");
      }
      catch (IOException e) {
         pass();
      }
   }
   public void testMultipleLines() throws IOException {
      writeln("line1");
      writeln("line2,line2");
      writeln("line3");
      CSVReader reader = getReaderAndCloseWriter();
      assertEquals(1, reader.next().size());
      assertEquals(2, reader.next().size());
      assertEquals(1, reader.next().size());
      assert(!reader.hasNext());
   }
   public void testComment() throws IOException {
      writeln("line1 data");
      writeln("# this is a comment");
      String line3data = "line3,some,data,columns";
      writeln(line3data);
      CSVReader reader = getReaderAndCloseWriter();
      reader.next();
      List line3Columns = reader.next();
      assertEquals("line3", line3Columns.get(0));
      assert(!reader.hasNext());
   }
   public void testMoreComments() throws IOException {
      writeln("# .... ok");
      writeln("# .... well?");
      CSVReader reader = getReaderAndCloseWriter();
      assert(!reader.hasNext());
   }
   public void testDoubleQuotedData() throws IOException {
      writeln("1015 Tenth Street,\"Laurel, MD 20707\", US");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(3, columns.size());
      assertEquals("1015 Tenth Street", columns.get(0));
      assertEquals("Laurel, MD 20707", columns.get(1));
      assertEquals("US", columns.get(2));
   }
   public void testMoreThanOneCommaInDoubleQuotedData()
      throws IOException {
      writeln("1015 Tenth Street,\"Laurel, MD 20707, US\"");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(2, columns.size());
      assertEquals("1015 Tenth Street", columns.get(0));
      assertEquals("Laurel, MD 20707, US", columns.get(1));
   }
   public void testEmbeddedQuotesArePartOfString() throws IOException {
      writeln("1015 Tenth Street, Jeff \"AC\" Langr, 1964");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(3, columns.size());
      assertEquals("1015 Tenth Street", columns.get(0));
      assertEquals("Jeff \"AC\" Langr", columns.get(1));
      assertEquals("1964", columns.get(2));
   }
   public void testSingleEmbeddedDoubleQuoteIsPartOfString()
      throws IOException {
      writeln("normally you wouldn\"t do this");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(1, columns.size());
      assertEquals("normally you wouldn\"t do this", columns.get(0));
   }
   public void testUnmatchedDoubleQuoteIsAnError() throws IOException {
      writeln("\"jkl");
      CSVReader reader = getReaderAndCloseWriter();
      try {
         reader.next();
         fail("should have thrown IO exception");
      }
      catch (IOException e)  {
         pass();
      }
   }
   public void testEmptyFields() throws IOException {
      writeln("");
      writeln(",");
      writeln(",a,,,");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(1, columns.size());
      assertEquals("", columns.get(0));
      columns = reader.next();
      assertEquals(2, columns.size());
      columns = reader.next();
      assertEquals(5, columns.size());
   }
   public void testTrim() throws IOException {
      writeln(
   "      this     ,       is     ,\tthe    end\t, \t of , it, all ");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(6, columns.size());
      assertEquals("this", columns.get(0));
      assertEquals("is", columns.get(1));
      assertEquals("the    end", columns.get(2));
      assertEquals("of", columns.get(3));
      assertEquals("it", columns.get(4));
      assertEquals("all", columns.get(5));
   }
   public void testNotTrim() throws IOException {
      writeln(
         "\"      this should be flanked by white space        \"");
      CSVReader reader = getReaderAndCloseWriter();
      List columns = reader.next();
      assertEquals(1, columns.size());
      assertEquals(
"      this should be flanked by white space        ", columns.get(0));
   }
   int writeln(String string) throws IOException {
      writer.write(string, 0, string.length());
      writer.write("\r\n", 0, 2);
      return string.length() + 2;
   }
   CSVReader getReaderAndCloseWriter() throws IOException {
      writer.close();
      return new CSVReader(filename);
   }
   final void pass() {}
   private String filename;
   private BufferedWriter writer;
}
```