

Refactoring Example

Written by Glen Stampoultzis

gstamp@iprimus.com.au

ICQ: 62722370

This example is intended to demonstrate how an ugly piece of code can be transformed using small steps until it reaches a state that is readable and maintainable. The example was actual code I wrote myself in a hurry. The purpose of this code is to take a string expression of the form:

Name='harry',Age='10'

and convert it into a map. Any invalid input should be converted to an `IllegalArgumentException`.

The critical principle behind refactoring is to refactor in the smallest steps you can manage. Working in large steps typically results in making too many mistakes and spending large amounts of time in a debugger. Small steps are great because you know at any point in time that there's only a handful of lines that could possibly be causing your tests to fail.

Before beginning any refactoring it is important to make sure you have tests to prove that the modifications you are doing aren't breaking the code. Tests should be run at the end of every refactoring step. If the changes you just made break the tests rollback to the last known good run and try again. This sounds rather harsh but small refactoring steps make this rollback process much easier.

The test cases I used to validate the code follows:

```
public class CriteriaTester
    extends TestCase
{
    public CriteriaTester(String s)
    {
        super(s);
    }

    public void testBasicCriteria()
        throws Exception
    {
        Criteria criteria = new Criteria("key='value'");
        Map values = criteria.getKeyValuePairs();

        assertEquals(new String[]{"key"}, values.keySet());
        assertEquals(new String[]{"value"}, values.values());
    }

    public void testMultipleCriteria()
        throws Exception
    {
        Criteria criteria = new Criteria("key1='value1',key2='value2'");
        Map values = criteria.getKeyValuePairs();

        assertEquals(new String[]{"key1", "key2"}, values.keySet());
        assertEquals(new String[]{"value1", "value2"}, values.values());
    }

    public void testWierdCriteria()
        throws Exception
    {
        Criteria criteria = new Criteria("key1='',='value2',aaaa='asd,vvv'");
        Map values = criteria.getKeyValuePairs();

        assertEquals(new String[]{"key1", "", "aaaa"}, values.keySet());
    }
}
```

```
    assertEquals(new String[]{"", "value2", "asd.vvv"}, values.values());
}

public void testFailingMatches()
{
    checkFail("asd");
    checkFail("asd=");
    checkFail("asd=");
    checkFail("asd=' '");
    checkFail(",asd=' '");
    checkFail(",=' '");
    checkFail("a=', 'a=");
}

public void testNull()
    throws Exception
{
    Criteria criteria = new Criteria(null);
    assertEquals(0, criteria.getKeyValuePairs().size());
}

private void checkFail(String criteria)
{
    try
    {
        new Criteria(criteria).getKeyValuePairs();
        fail("Criteria incorrectly passed: " + criteria);
    } catch (IllegalArgumentException e)
    {
    }
}

public void assertEquals(Object[] actual, Collection expected)
{
    assertTrue(actual.length == expected.size());
    for (int i = 0; i < actual.length; i++)
    {
        Object value = actual[i];
        assertTrue(expected.contains(value));
    }
}
}
```

The code we are starting with is a great example of poorly structured hard to read code. We all have our bad days I guess.

My initial changes are simple. I generally start by performing the extract method refactoring. Extract method works well as a first refactoring because it works well in breaking down a big block of code into something more readable. Sometimes extract method is not practical because there are too many local variables. In this case consider creating a method object.

Often it is necessary to adjust the scope of some variables before starting with extract method as you will see further on.

Using extract method is a good choice on the code below because it allows me to better see the structure of the loop.

```
public class Criteria
{
    private String criteria;
    int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                int equalsIndex = criteria.indexOf('=', currentPosition);
                if (equalsIndex == -1)
                    throw new IllegalArgumentException("Expected '=' character");
                if (criteria.charAt(equalsIndex + 1) != '\\')
                    throw new IllegalArgumentException("Expected single quote after equals character");
                int nextQuote = criteria.indexOf('\\'', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String key = criteria.substring(currentPosition, equalsIndex);
                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.length() > 0 && criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }
}
```

Before I perform an extract method I notice a redundant clause in the if statement after the while loop. It is checking the length of criteria string. This is not necessary however; the check is already done at the start of the method. It is a simple matter to remove it.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                int equalsIndex = criteria.indexOf('=', currentPosition);
                if (equalsIndex == -1)
                    throw new IllegalArgumentException("Expected '=' character");
                if (criteria.charAt(equalsIndex + 1) != '\\')
                    throw new IllegalArgumentException("Expected single quote after equals character");
                int nextQuote = criteria.indexOf('\\', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String key = criteria.substring(currentPosition, equalsIndex);
                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }
}
```

I start looking for methods to extract but run into a slight problem; multiple fields are being modified in the body of the loop. Before I can start extracting any methods I need to convert the local variable *equalsIndex* to a field.

```
public class Criteria
{
    private String criteria;
    int currentPosition = 0;
    private int equalsIndex;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                equalsIndex = criteria.indexOf('=', currentPosition);
                if (equalsIndex == -1)
                    throw new IllegalArgumentException("Expected '=' character");
                if (criteria.charAt(equalsIndex + 1) != '\\')
                    throw new IllegalArgumentException("Expected single quote after equals character");
                int nextQuote = criteria.indexOf('\\', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String key = criteria.substring(currentPosition, equalsIndex);
                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }
}
```

Now that the *equalsIndex* field has been modified I'm free to extract the code that parses the key part from the string. This code is actually non-contiguous. If you do need to extract non-contiguous segments of code be careful of side effects. In this case I don't have any.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                if (criteria.charAt(equalsIndex + 1) != '\\')
                    throw new IllegalArgumentException("Expected single quote after equals character");
                int nextQuote = criteria.indexOf('\\', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

Next I extract the code that matches the quote. Nothing complicated here. There are no parameters needed and no return values.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                matchSingleQuote();
                int nextQuote = criteria.indexOf('\'', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\'',')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\'',')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

Again to extract the next segment of code I need to convert one of the locals to a field.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                matchSingleQuote();
                nextQuote = criteria.indexOf('\'', equalsIndex + 2);
                if (nextQuote == -1)
                    throw new IllegalArgumentException("No closing single quote found");

                String value = criteria.substring(equalsIndex + 2, nextQuote);

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\'',')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\'',')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

With the local taken care of I can convert the code that parses the value part into it's own method.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                matchSingleQuote();
                String value = matchUntilSingleQuote();

                if (key.indexOf(',') >= 0)
                    throw new IllegalArgumentException("Key can not contain comma");

                result.put(key, value);

                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\', equalsIndex + 2);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(equalsIndex + 2, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

In the next step I extract the key validation code into it's own method. At the same time I've moved the validation code after the key matching code since it seems more logical this way.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                validateKey(key);
                matchSingleQuote();
                String value = matchUntilSingleQuote();

                result.put(key, value);

                currentPosition = nextQuote + 1;
                if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
                    throw new IllegalArgumentException("Expected comma");
                currentPosition++;
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\'', equalsIndex + 2);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(equalsIndex + 2, nextQuote);
        return value;
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

Next comes yet another extract method refactoring. Luckily my IDE (IntelliJ Idea) provides automated support for this or it could easily become tedious and a lot slower.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                validateKey(key);
                matchSingleQuote();
                String value = matchUntilSingleQuote();

                result.put(key, value);

                matchCommaOrEndOfLine();
            }
            if (criteria.charAt(criteria.length()-1) != '\\')
                throw new IllegalArgumentException("Expecting comma");

            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\'', equalsIndex + 2);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(equalsIndex + 2, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

And yet another extract method refactoring...

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        try
        {
            while (currentPosition < criteria.length())
            {
                String key = matchUntilEqualSign();
                validateKey(key);
                matchSingleQuote();
                String value = matchUntilSingleQuote();
                result.put(key, value);
                matchCommaOrEndOfLine();
            }
            checkNoCommaAtEndOfLine();
            return result;
        } catch (StringIndexOutOfBoundsException e)
        {
            throw new IllegalArgumentException("Invalid criteria");
        }
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length()-1) != '\\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\'', equalsIndex + 2);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(equalsIndex + 2, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (criteria.charAt(equalsIndex + 1) != '\\\'')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

We're now at a point where the loop itself reads more like a bunch of comments. There's still quite a lot of refactoring that still need to be done however.

The next thing that catches my eye is the ugly catch block. This is a really poor way to check a condition. I replace it with an explicit check in *matchSingleQuote()*.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int equalsIndex;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntilEqualSign();
            validateKey(key);
            matchSingleQuote();
            String value = matchUntilSingleQuote();
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\'', equalsIndex + 2);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(equalsIndex + 2, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (equalsIndex + 1 >= criteria.length() || criteria.charAt(equalsIndex + 1) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        return key;
    }
}
```

Notice *matchUntilSingleQuote()* and *matchUntilEqualSign()* are very similar. The main problem is the indexes being used. Looking closely *equalsIndex* and *nextQuote* are not really needed. It should be possible to combine them into *currentPosition*. Consolidating independent variables like this should be done with caution. It should only be done when the meaning of the variables is the same and the usage of one variable naturally flows onto the next.

First I remove *equalsIndex*. I do this by identifying where it's altered and changing it to a local. Since it originally started out as a local this is somewhat ironic but not really uncommon. In the next step I added a line to update *currentPosition* with the new value of *equalsIndex*. I cheat a little and adjust *currentPosition* to be *equalsIndex* plus one so that *currentPosition* represents the true current position. This requires adjusting the existing references to *equalsIndex*.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;
    private int nextQuote;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntilEqualSign();
            validateKey(key);
            matchSingleQuote();
            String value = matchUntilSingleQuote();
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        nextQuote = criteria.indexOf('\\', currentPosition + 1);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(currentPosition + 1, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        int equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
    }
}
```

```
        currentPosition = equalsIndex + 1;
        return key;
    }
}
```

Next its time to eliminate *nextQuote*. Lets make *currentPosition* private while we're at it.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntilEqualSign();
            validateKey(key);
            matchSingleQuote();
            String value = matchUntilSingleQuote();
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private String matchUntilSingleQuote()
    {
        int nextQuote = criteria.indexOf('\\'', currentPosition + 1);
        if (nextQuote == -1)
            throw new IllegalArgumentException("No closing single quote found");

        String value = criteria.substring(currentPosition + 1, nextQuote);
        currentPosition = nextQuote + 1;
        return value;
    }

    private void matchSingleQuote()
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != '\\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntilEqualSign()
    {
        int equalsIndex = criteria.indexOf('=', currentPosition);
        if (equalsIndex == -1)
            throw new IllegalArgumentException("Expected '=' character");
        String key = criteria.substring(currentPosition, equalsIndex);
        currentPosition = equalsIndex + 1;
        return key;
    }
}
```

Okay, *matchUntilSingleQuote()* and *matchUntilEqualSign()* are pretty much identical now. The only difference is the character being matched and the exception message being thrown. The message is not terribly important so it can be altered without affecting the clients of the class. To combine the two methods I need to add a parameter to one of the methods and rename it to something more appropriate.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntil('=');
            validateKey(key);
            matchSingleQuote();
            String value = matchUntil('\');
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchSingleQuote()
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != '\')
            throw new IllegalArgumentException("Expected single quote after equals character");
    }

    private String matchUntil(char matchChar)
    {
        int charIndex = criteria.indexOf(matchChar, currentPosition);
        if (charIndex == -1)
            throw new IllegalArgumentException("Expected '" + matchChar + "' character");
        String result = criteria.substring(currentPosition, charIndex);
        currentPosition = charIndex + 1;
        return result;
    }
}
```

Running this my tests fail. Obviously I've missed something. I revert back to my previous refactoring and take a closer look at the two methods I have joined. I see one of them uses *currentPosition + 1* while the other simply uses *currentPosition*. This is because *matchStringQuote()* does not advance the *currentPosition* index. I fix this and rerun my tests. All passing.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntil('=');
            validateKey(key);
            matchSingleQuote();
            String value = matchUntil('\');
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchSingleQuote()
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != '\')
            throw new IllegalArgumentException("Expected single quote after equals character");
        currentPosition++;
    }

    private String matchUntil(char matchChar)
    {
        int charIndex = criteria.indexOf(matchChar, currentPosition);
        if (charIndex == -1)
            throw new IllegalArgumentException("Expected '" + matchChar + "' character");
        String result = criteria.substring(currentPosition, charIndex);
        currentPosition = charIndex + 1;
        return result;
    }
}
```

Next my attention falls on *matchCommaOrEndOfLine()*. I really don't like this method, it grates that I'm doing a double check at the end of the loop to ensure I didn't match a comma as the last character. If I move the check up to the start of the loop and don't do the check if we are at the start of the matching process that would mean I could rid myself of the check at the end of the loop. Furthermore I can see the possibility of making *matchSingleQuote()* more generic and using it for comma matching.

First things first, I add a parameter to *matchSingleQuote()* and rename it to *matchChar()*. I also alter the implementation slightly to match the parameter I'm passing.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
            matchCommaOrEndOfLine();
        }
        checkNoCommaAtEndOfLine();

        return result;
    }

    private void checkNoCommaAtEndOfLine()
    {
        if (criteria.charAt(criteria.length() - 1) != '\')
            throw new IllegalArgumentException("Expecting comma");
    }

    private void matchCommaOrEndOfLine()
    {
        if (currentPosition < criteria.length() && criteria.charAt(currentPosition) != ',')
            throw new IllegalArgumentException("Expected comma");
        currentPosition++;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
        currentPosition++;
    }

    private String matchUntil(char matchChar)
    {
        int charIndex = criteria.indexOf(matchChar, currentPosition);
        if (charIndex == -1)
            throw new IllegalArgumentException("Expected '" + matchChar + "' character");
        String result = criteria.substring(currentPosition, charIndex);
        currentPosition = charIndex + 1;
        return result;
    }
}
```

Now to reorganize the loop slightly.

```
public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || criteria.equals(""))
            return Collections.EMPTY_MAP;

        while (currentPosition < criteria.length())
        {
            if (currentPosition != 0)
                matchChar(',');
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
        }

        return result;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
    {
        if (currentPosition >= criteria.length() || criteria.charAt(currentPosition) != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
        currentPosition++;
    }

    private String matchUntil(char matchChar)
    {
        int charIndex = criteria.indexOf(matchChar, currentPosition);
        if (charIndex == -1)
            throw new IllegalArgumentException("Expected '" + matchChar + "' character");
        String result = criteria.substring(currentPosition, charIndex);
        currentPosition = charIndex + 1;
        return result;
    }
}
```

At this point in the code I was reasonably happy with the code but a friend suggested it was still not finished. Talking with him we agreed that the implementation probably needed to use a stream.

I was stuck for a long time at this point in the code. I knew pretty much that string traversal would work better as a stream but I wasn't sure how to get from the current code to a stream in small steps. Then it came to me. I was facing two problems. The first was that I still had duplication in the code. The second was that I needed to refactor to make room for change to a stream first. What I mean by making room is that I need to refactor so that the code more closely matches the operations you would perform when reading from a stream.

The duplication was not immediately apparent to me because it was very fine grained and present in three separate forms. They are:

- `criteria.equals("")`
- `currentPosition < criteria.length()`
- `currentPosition >= criteria.length()`

The third form of duplication is simply the negation of the second.

```

public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        while (beforeEnd())
        {
            if (currentPosition != 0)
                matchChar(',');
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
        }

        return result;
    }

    private boolean beforeEnd()
    {
        return currentPosition < criteria.length();
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
    {
        if (!beforeEnd() || criteria.charAt(currentPosition) != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
        currentPosition++;
    }

    private String matchUntil(char matchChar)
    {
        int charIndex = criteria.indexOf(matchChar, currentPosition);
        if (charIndex == -1)
            throw new IllegalArgumentException("Expected '" + matchChar + "' character");
        String result = criteria.substring(currentPosition, charIndex);
        currentPosition = charIndex + 1;
        return result;
    }
}

```

Now that the duplication is gone I need to make room for the stream. One of the issues that we face is that there are functions like *indexOf()* and *substring()* that don't have direct equivalents in a stream. Lets use the substitute algorithm refactoring on *matchUntil()* and remove these functions:

```

public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        while (beforeEnd())
        {
            if (currentPosition != 0)
                matchChar(',');
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
        }
    }
}

```

```

    return result;
}

private boolean beforeEnd()
{
    return currentPosition < criteria.length();
}

private void validateKey(String key)
{
    if (key.indexOf(',') >= 0)
        throw new IllegalArgumentException("Key can not contain comma");
}

private void matchChar(int matchChar)
{
    if (!beforeEnd() || criteria.charAt(currentPosition) != matchChar)
        throw new IllegalArgumentException("Expected '" + matchChar + "'");
    currentPosition++;
}

private String matchUntil(char matchChar)
{
    StringBuffer result = new StringBuffer();
    while (beforeEnd())
    {
        if (criteria.charAt(currentPosition) == matchChar)
        {
            currentPosition++;
            return result.toString();
        }
        result.append(criteria.charAt(currentPosition++));
    }

    throw new IllegalArgumentException("Expected '" + matchChar + "' character");
}
}

```

By doing this we've created some further duplication. Lets remove it.

```

public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        while (beforeEnd())
        {
            if (currentPosition != 0)
                matchChar(',');
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
        }

        return result;
    }

    private boolean beforeEnd()
    {
        return currentPosition < criteria.length();
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
    {
        if (!beforeEnd() || nextChar() != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
    }

    private String matchUntil(char matchChar)
    {
        StringBuffer result = new StringBuffer();
        while (beforeEnd())
        {
            char c = nextChar();
            if (c == matchChar)
                return result.toString();
            result.append(c);
        }
    }
}

```

```

    }
    throw new IllegalArgumentException("Expected '" + matchChar + "' character");
}

private char nextChar()
{
    return criteria.charAt(currentPosition++);
}
}

```

Okay, so now we have isolated the code that deals with strings into two very small one-line routines. There is still the issue of *currentPosition* being used in the main loop. This is really a bad use of this variable anyway. We really should introduce a variable specifically for the purpose of telling whether this is the first run through the loop.

```

public class Criteria
{
    private String criteria;
    private int currentPosition = 0;

    public Criteria(String criteria)
    {
        this.criteria = criteria;
    }

    public Map getKeyValuePairs()
    {
        Map result = new HashMap();

        if (criteria == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        boolean firstRunThroughLoop = true;
        while (beforeEnd())
        {
            if (!firstRunThroughLoop)
                matchChar(',');
            String key = matchUntil('=');
            validateKey(key);
            matchChar('\');
            String value = matchUntil('\');
            result.put(key, value);
            firstRunThroughLoop = false;
        }

        return result;
    }

    private boolean beforeEnd()
    {
        return currentPosition < criteria.length();
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
    {
        if (!beforeEnd() || nextChar() != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
    }

    private String matchUntil(char matchChar)
    {
        StringBuffer result = new StringBuffer();
        while (beforeEnd())
        {
            char c = nextChar();
            if (c == matchChar)
                return result.toString();
            result.append(c);
        }

        throw new IllegalArgumentException("Expected '" + matchChar + "' character");
    }

    private char nextChar()
    {
        return criteria.charAt(currentPosition++);
    }
}

```

Although this new variable makes it easier to understand what's going on, I still feel it would be even easier if I renamed it. So I do.

```

public class Criteria
{

```

```
private String criteria;
private int currentPosition = 0;

public Criteria(String criteria)
{
    this.criteria = criteria;
}

public Map getKeyValuePairs()
{
    Map result = new HashMap();

    if (criteria == null || !beforeEnd())
        return Collections.EMPTY_MAP;

    boolean skipComma = true;
    while (beforeEnd())
    {
        if (!skipComma)
            matchChar(',');
        String key = matchUntil('=');
        validateKey(key);
        matchChar('\n');
        String value = matchUntil('\n');
        result.put(key, value);
        skipComma = false;
    }

    return result;
}

private boolean beforeEnd()
{
    return currentPosition < criteria.length();
}

private void validateKey(String key)
{
    if (key.indexOf(',') >= 0)
        throw new IllegalArgumentException("Key can not contain comma");
}

private void matchChar(int matchChar)
{
    if (!beforeEnd() || nextChar() != matchChar)
        throw new IllegalArgumentException("Expected '" + matchChar + "'");
}

private String matchUntil(char matchChar)
{
    StringBuffer result = new StringBuffer();
    while (beforeEnd())
    {
        char c = nextChar();
        if (c == matchChar)
            return result.toString();
        result.append(c);
    }

    throw new IllegalArgumentException("Expected '" + matchChar + "' character");
}

private char nextChar()
{
    return criteria.charAt(currentPosition++);
}
}
```

Now that *currentPosition* is also isolated, migrating to a stream is much easier. There is, however, one complication. Readers don't provide a way to determine whether the end of stream has been reached until you attempt to read beyond the last character. This causes a problem for the code as it currently stands as it assumes it's possible to determine whether there are more characters to read at any point. To get around this I changed the system to read ahead one character.

There are two possible alternatives that I might have taken. I could have used *mark()* and *reset()* to read forward one character. This method is not supported on all Readers however. Another alternative approach would have been to try to refactor the code so that it no longer relies on knowing whether it's at the last character of the string. That might have been a little tricky.

```
public class Criteria
{
    private Reader criteriaReader;
    private int lastChar;

    public Criteria(String criteria)
```

```

throws Exception
{
    if (criteria == null)
        this.criteriaReader = null;
    else
    {
        this.criteriaReader = new StringReader(criteria);
        lastChar = criteriaReader.read();
    }
}

public Map getKeyValuePairs()
    throws IOException
{
    Map result = new HashMap();

    if (criteriaReader == null || !beforeEnd())
        return Collections.EMPTY_MAP;

    boolean skipComma = true;
    while (beforeEnd())
    {
        if (!skipComma)
            matchChar(',');
        String key = matchUntil('=');
        validateKey(key);
        matchChar('\n');
        String value = matchUntil('\n');
        result.put(key, value);
        skipComma = false;
    }

    return result;
}

private boolean beforeEnd()
{
    return lastChar != -1;
}

private void validateKey(String key)
{
    if (key.indexOf(',') >= 0)
        throw new IllegalArgumentException("Key can not contain comma");
}

private void matchChar(int matchChar)
    throws IOException
{
    if (!beforeEnd() || nextChar() != matchChar)
        throw new IllegalArgumentException("Expected '" + matchChar + "'");
}

private String matchUntil(char matchChar)
    throws IOException
{
    StringBuffer result = new StringBuffer();
    while (beforeEnd())
    {
        char c = nextChar();
        if (c == matchChar)
            return result.toString();
        result.append(c);
    }

    throw new IllegalArgumentException("Expected '" + matchChar + "' character");
}

private char nextChar()
    throws IOException
{
    char result = (char)lastChar;
    lastChar = criteriaReader.read();
    return result;
}
}

```

I was tempted to stop at this point but someone suggested that the loop in *getKeyValuePairs()* was not clearly showing my intent. They were right. A few more changes need to be made. Let's extract the lines of code that match the key and the value.

```

public class Criteria
{
    private Reader criteriaReader;
    private int lastChar;

    public Criteria(String criteria)
        throws Exception
    {
        if (criteria == null)
            this.criteriaReader = null;
        else
        {
            this.criteriaReader = new StringReader(criteria);
            lastChar = criteriaReader.read();
        }
    }
}

```

```

    }
}

public Map getKeyValuePairs()
    throws IOException
{
    Map result = new HashMap();

    if (criteriaReader == null || !beforeEnd())
        return Collections.EMPTY_MAP;

    boolean skipComma = true;
    while (beforeEnd())
    {
        if (!skipComma)
            matchChar(',');
        String key = matchKey();
        String value = matchValue();
        result.put(key, value);
        skipComma = false;
    }

    return result;
}

private String matchValue() throws IOException
{
    matchChar('\');
    String value = matchUntil('\');
    return value;
}

private String matchKey() throws IOException
{
    String key = matchUntil('=');
    validateKey(key);
    return key;
}

private boolean beforeEnd()
{
    return lastChar != -1;
}

private void validateKey(String key)
{
    if (key.indexOf(',') >= 0)
        throw new IllegalArgumentException("Key can not contain comma");
}

private void matchChar(int matchChar)
    throws IOException
{
    if (!beforeEnd() || nextChar() != matchChar)
        throw new IllegalArgumentException("Expected '" + matchChar + "'");
}

private String matchUntil(char matchChar)
    throws IOException
{
    StringBuffer result = new StringBuffer();
    while (beforeEnd())
    {
        char c = nextChar();
        if (c == matchChar)
            return result.toString();
        result.append(c);
    }

    throw new IllegalArgumentException("Expected '" + matchChar + "' character");
}

private char nextChar()
    throws IOException
{
    char result = (char)lastChar;
    lastChar = criteriaReader.read();
    return result;
}
}
}

```

Better but the loop is still made a little ugly by the comma checking code. As a first step towards cleaning that up I'm going to change from a while loop into a do..while loop. I can do this because I know it always runs through the at least once. Running the tests will help confirm this assertion.

```

public class Criteria
{
    private Reader criteriaReader;
    private int lastChar;

    public Criteria(String criteria)
        throws Exception
    {
        if (criteria == null)
            this.criteriaReader = null;
    }
}

```

```

        else
        {
            this.criteriaReader = new StringReader(criteria);
            lastChar = criteriaReader.read();
        }
    }

    public Map getKeyValuePairs()
        throws IOException
    {
        Map result = new HashMap();

        if (criteriaReader == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        boolean skipComma = true;
        do
        {
            if (!skipComma)
                matchChar(',');
            String key = matchKey();
            String value = matchValue();
            result.put(key, value);
            skipComma = false;
        } while (beforeEnd());

        return result;
    }

    private String matchValue() throws IOException
    {
        matchChar('\n');
        String value = matchUntil('\n');
        return value;
    }

    private String matchKey() throws IOException
    {
        String key = matchUntil('=');
        validateKey(key);
        return key;
    }

    private boolean beforeEnd()
    {
        return lastChar != -1;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
        throws IOException
    {
        if (!beforeEnd() || nextChar() != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
    }

    private String matchUntil(char matchChar)
        throws IOException
    {
        StringBuffer result = new StringBuffer();
        while (beforeEnd())
        {
            char c = nextChar();
            if (c == matchChar)
                return result.toString();
            result.append(c);
        }

        throw new IllegalArgumentException("Expected '" + matchChar + "' character");
    }

    private char nextChar()
        throws IOException
    {
        char result = (char)lastChar;
        lastChar = criteriaReader.read();
        return result;
    }
}

```

Now to simplify the comma matching part of the loop I create a new method and take advantage of short circuit boolean evaluation.

```

public class Criteria
{
    private Reader criteriaReader;
    private int lastChar;

    public Criteria(String criteria)
        throws Exception
    {
        if (criteria == null)
            this.criteriaReader = null;
    }
}

```

```

        else
        {
            this.criteriaReader = new StringReader(criteria);
            lastChar = criteriaReader.read();
        }
    }

    public Map getKeyValuePairs()
        throws IOException
    {
        Map result = new HashMap();

        if (criteriaReader == null || !beforeEnd())
            return Collections.EMPTY_MAP;

        do
        {
            String key = matchKey();
            String value = matchValue();
            result.put(key, value);
        } while (beforeEnd() && matchComma());

        return result;
    }

    private boolean matchComma() throws IOException
    {
        matchChar(',');
        return true;
    }

    private String matchValue() throws IOException
    {
        matchChar('\');
        String value = matchUntil('\');
        return value;
    }

    private String matchKey() throws IOException
    {
        String key = matchUntil('=');
        validateKey(key);
        return key;
    }

    private boolean beforeEnd()
    {
        return lastChar != -1;
    }

    private void validateKey(String key)
    {
        if (key.indexOf(',') >= 0)
            throw new IllegalArgumentException("Key can not contain comma");
    }

    private void matchChar(int matchChar)
        throws IOException
    {
        if (!beforeEnd() || nextChar() != matchChar)
            throw new IllegalArgumentException("Expected '" + matchChar + "'");
    }

    private String matchUntil(char matchChar)
        throws IOException
    {
        StringBuffer result = new StringBuffer();
        while (beforeEnd())
        {
            char c = nextChar();
            if (c == matchChar)
                return result.toString();
            result.append(c);
        }

        throw new IllegalArgumentException("Expected '" + matchChar + "' character");
    }

    private char nextChar()
        throws IOException
    {
        char result = (char)lastChar;
        lastChar = criteriaReader.read();
        return result;
    }
}

```

Now it's absolutely clear what's going on. We match a key, then a value and put the results in a map. There are a lot of methods to get your head around but they are each small and relatively easily understood. Of course there are many ways we could have approached this refactoring and the end result may not have looked anything like what we have now. The important point is that we were able to make our changes in small steps and still have working code after each step. This is an extremely important point. Large changes introduce risk and increase the likelihood of failure. With small

changes we don't have to wait long to see that the change we had in mind actually work. This means that we can stop at any stage.

I'd like to thank all the reviewers from the Melbourne XP group. Your help was greatly appreciated.