xp123.com

The Test/Code Cycle in XP, Part 1: Model

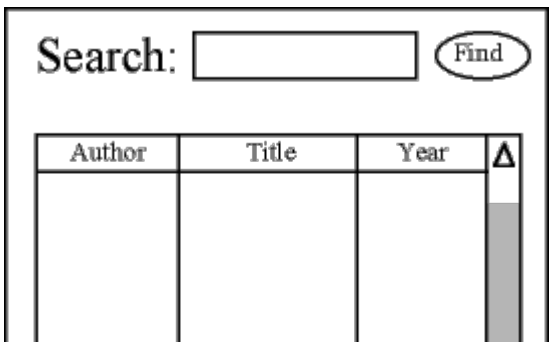
Catch Themes

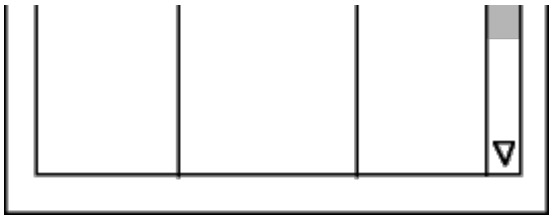
18-23 Minuten

This paper demonstrates the development of a small bibliographic system using Extreme Programming techniques. Part 1 shows development of the model; part 2 shows development of the associated user interface.

Specifically, it shows how unit tests and simple design work together while programming. Observe how the coding process occurs in small steps, just enough implementation to make each test run. There's a rhythm to it, like a pendulum of a clock: test a little, code a little, test a little, code a little. (To bring this out, we'll align test code to the left of the page, and application code to the right.)

For this example, suppose we have bibliographic data with author, title, and year of publication. Our goal is to write a system that can search that information for values we specify. We have in mind an interface something like this:





We'll divide our work into two parts: the model and the user interface.

For our model, we have a collection of Documents, which know their metadata. A Searcher knows how to find Documents: given a Query, it will return a Result (a set of Documents). We'll create our unit tests (and our classes) bottom-up: Document, Result, Query, and Searcher.

Document, Result, and Query

Document

A **Document** needs to know its author, title, and year. We'll start with a "data bag" classs, beginning with its test:

```
public void testDocument() {
    Document d = new
Document("a", "t", "y");
    assertEquals("a",
d.getAuthor());
    assertEquals("t",
d.getTitle());
    assertEquals("y",
d.getYear());
}
```

This test doesn't compile (as we haven't created the Document class yet). Create the class with stubbed-out methods.

Run the test again to make sure it fails. This may seem funny – don't we want the tests to pass? Yes, we do. But by seeing them fail first, we get some assurance that the test is valid. And once in a while, a test passes unexpectedly: "that's interesting!"

Fill in the constructor and the methods to make the test pass.

Let's highlight this mini-process:

The Test/Code Cycle in XP

- Write one test.
- Compile the test. It should fail, as you haven't implemented anything yet.
- Implement just enough to compile. (Refactor first if necessary.)
- Run the test and see it fail.
- Implement just enough to make the test pass.
- Run the test and see it pass.
- Refactor for clarity and "once and only once".
- Repeat from the top.

This process ensures that you've seen the test both fail and pass, which gives you assurance that the test did test something, that your change made a difference, and that

you've added valued functionality.

Some people will advocate not bothering to test simple setter and getter methods. ("They can't possibly be wrong.", "It's a pain to write a bunch of getter/setter tests.") I tend to write the tests anyway:

- It doesn't take *that* much time to write the test, and it's certainly not hard, but it gives you that extra edge. ("You thought you were sure it's ok; now you have a test that demonstrates it.")
- A test will often have a longer lifetime than the code it's testing. The test is there so when you add caching, or don't create objects until required, or add logging, etc., you still have assurance that the original function will work.
- Boring tests full of setters and getters are often trying to tell you something: the class may not be pulling its weight. When a class is almost a "struct", it's often a sign that the responsibilities aren't distributed right between classes.

Result

A **Result** needs to know two things: the total number of items, and the list of Documents it contains. First we'll test that an empty result has no items.

```
public void testEmptyResult() {
    Result r = new Result();
    assert ("count=0 for empty
result", r.getCount() == 0);
}
```

Create the Result class and stub out its `getCount()`

method. See it fail until you add "`return 0;`" as its implementation.

Next test a result with two documents.

```
public void
testResultWithTwoDocuments() {
    Document d1 = new
Document("a1", "t1", "y1");
    Document d2 = new
Document("a2", "t2", "y2");
    Result r = new Result(new
Document[]{d1, d2});
    assert (r.getCount() == 2);
    assert (r.getItem(0) == d1);
    assert (r.getItem(1) == d2);
}
```

Add the `getItem()` method (returning null) and watch the test fail. (I'm going to stop mentioning that, but keep doing it. It takes a few seconds, but gives that extra bit of reassurance.) Implementing a simple version of Result will give:

```
public class Result {
    Document[] collection = new
Document[0];

    public Result() {}

    public
Result(Document[]collection) {
        this.collection =
```

```
collection;
    }

    public int getCount()
{return collection.length;}

    public Document getItem(int
i) {return collection[i];}
}
```

The test runs, so we're done with this class.

Query

We'll represent the **Query** as just its query string.

```
public void testSimpleQuery() {
    Query q = new Query("test");
    assertEquals("test",
q.getValue());
}
```

Create the Query class with a constructor, so that it remembers its query string and reports it via `getValue()`.

Searcher

The **Searcher** is the most interesting class. The easy case is first: we should get nothing back from an empty collection of Documents.

```
public void
testEmptyCollection() {
    Searcher searcher = new
```

```
Searcher();
    Result r = searcher.find(new
Query("any"));
    assert(r.getCount() == 0);
}
```

This test doesn't compile, so stub out the Searcher class.

```
public class Searcher {
    public Searcher() {}
    Result find(Query q) {return
null;}
}
```

The test compiles, but fails to run correctly (because `find()` returns null). We can fix this with this change: "`public Result find(Query q) {return new Result();}`".

Things get more interesting when we try real searches. Then we face the issue of where the Searcher gets its documents. We'll begin by passing an array of Documents to the Searcher's constructor. But first, a test.

```
public void
testOneElementCollection() {
    Document d = new
Document("a", "a word here",
"y");
    Searcher searcher = new
Searcher(new Document[]{d});

    Query q1 = new
```

```
Query("word");
    Result r1 =
searcher.find(q1);
    assert(r1.getCount() == 1);

    Query q2 = new
Query("notThere");
    Result r2 =
searcher.find(q2);
    assert (r2.getCount() == 0);
}
```

This test shows us that we have to find what *is* there, and not find what's *not* there.

To implement this, we have to provide the new constructor that makes the test compile (though it still fails). Then we have to get serious about implementation.

First, we can see that a search has to retain knowledge of its collection between calls to `find()`, so we'll add a member variable to keep track, and have the constructor remember its argument:

```
Document[] collection = new
Document[0];

public Searcher(Document[] docs)
{
    this.collection = docs;
}
```

Now, the simplest version of `find()` can iterate through its documents, adding each one that matches the query to a

Result:

```
public Result find(Query q) {
    Result result = new
Result();
    for (int i = 0; i <
collection.count; i++) {
        if
(collection[i].matches(q)) {

result.add(collection[i]);
        }
    }
    return result;
}
```

This looks good, except for two problems: Document has no `matches()` method, and Result has no `add()` method.

Let's add a test: we'll check that each field can be matched, and that a document doesn't match queries it shouldn't:

```
public void
testDocumentMatchingQuery() {
    Document d = new
Document("1a", "t2t", "y3");
    assert(d.matches(new
Query("1")));
    assert(d.matches(new
Query("2")));
    assert(d.matches(new
Query("3")));
    assert(!d.matches(new
```

```
Query("4")));
}
```

There are three situations for queries that we should deal with eventually: empty queries, partial matches, and case sensitivity. For now, we'll assume empty strings and partial matches should match, and the search is case-sensitive. In the future we might change our mind.

This is enough information to let us implement matches:

```
public boolean matches(Query q)
{
    String query = q.getValue();

    return
       author.indexOf(query) !=
-1
    || title.indexOf(query) !=
-1
    || year.indexOf(query) !=
-1;
}
```

This will enable `testDocumentMatchingQuery()` to work, but `testOneElementCollection()` will still fail, because Result has no `add()` method yet. So, add a test for the method `Result.add()`:

```
public void testAddingToResult()
{
    Document d1 = new
StringDocument("a1", "t1",
```

```
"y1");
    Document d2 = new
StringDocument("a2", "t2",
"y2");

    StringResult r = new
StringResult();
    r.add(d1);
    r.add(d2);

    assert ("2 items in result",
r.getCount() == 2);
    assert ("First item",
r.getItem(0) == d1);
    assert ("Second item",
r.getItem(1) == d2);
}
```

This test fails. Result already remembers its list by using an array, but that is not the best choice for a structure that needs to change its size. We'll change to use a Vector:

```
Vector collection = new Vector();

public Result(Document[] docs) {
    for (int i = 0; i < docs.length;
i++) {

this.collection.addElement(docs[i]);
    }
}
```

```
public int getCount() {return
collection.size();}

public Document getItem(int i) {
    return
(Document)collection.elementAt(i);
}
```

Make sure the old unit tests `testEmptyResult()` and `testResultWithTwoDocuments()` still pass. Add the new method:

```
public void add(Document d) {
    collection.addElement(d);
}
```

Let's consider the the `new Result(Document[])` constructor. It was introduced to support the `testResultWithTwoDocuments()` test, because it was the only way we could create Results containing documents. Later, we introduced `Result.add()`, which is what the Searcher needs. The array constructor is no longer needed. So, we'll put on a testing hat and revise that test. Instead of `Result r = new Result(new Document[]{d1,d2});`, we'll use:

```
Result r = new Result();
r.add(d1);
r.add(d2);
```

We verify that all tests still pass, so it is now safe to remove the array-based constructor. We also see that

`testAddingToResult()` is now essentially a duplicate of `testResultWithTwoDocuments()`, so we'll remove the latter.

Finally, all our tests pass for Document, Result, Query, and Searcher.

Initialization

Loading Documents

Where does a searcher get its documents? Currently, you'd call its constructor from the main routine, passing in an array of documents. Instead, we want the searcher to own the process of loading its documents.

We begin with a test. We'll pass in a Reader, and be prepared to see exceptions. We've also postulated a `getCount()` method, used only by tests to verify that something was loaded. An advantage of having the tests in the same package as the class under test is that you can provide non-public methods that let tests view an object's internal state.

```
public void
testLoadingSearcher() {
    try {
        String docs =
"a1tt1ty1na2tt2ty2"; // t=field,
n=row
        StringReader reader =
new StringReader(docs);
            Searcher searcher =
```

```
new Searcher();

searcher.load(reader);
            assert("Loaded",
searcher.getCount() == 2);
    } catch (IOException e) {
        fail ("Loading
exception: " + e);
    }
}
```

Notice that Searcher still uses an array (the simplest choice at the time). We'll do as we did for Result, a refactoring converting from an array to a Vector.

```
package search;

import java.util.*;

public class Searcher {
    Vector collection = new
Vector();

    public Searcher() {}

    public Searcher(Document[]
docs) {
        for (int i = 0; i <
docs.length; i++) {

collection.addElement(docs[i]);
```

```
        }
    }

    public Query makeQuery(String
s) {
        return new Query(s);
    }

    public Result find(Query q) {
        Result result = new
Result();
        for (int i = 0; i <
collection.size(); i++) {
            Document doc =
(Document)collection.elementAt(i);
            if (doc.matches(q)) {
                result.add(doc);
            }
        }
        return result;
    }
}
```

(Verify that the old tests pass.) Now we're in a position to do the loading:

```
// Searcher:
public void load(Reader reader)
throws IOException {
    BufferedReader in = new
BufferedReader(reader);
```

```
    try {
        String line =
in.readLine();
        while (line != null) {

collection.addElement(new
Document(line));
            line =
in.readLine();
        }
    } finally {
        try {in.close();} catch
(Exception ignored) {}
    }
}

int getCount() {
    return collection.size();
}
```

```
// Document:
public Document(String line) {
    StringTokenizer tokens = new
StringTokenizer(line, "t");
    author = tokens.nextToken();
    title = tokens.nextToken();
    year = tokens.nextToken();
}
```

Searcher's array-based constructor is no longer needed. We'll adjust the test and delete the constructor:

```
public void
testOneElementCollection() {
    Searcher searcher = new
Searcher();
    try {
        StringReader reader =
new StringReader("ata word
herety");
        searcher.load(reader);
    } catch (Exception ex) {
        fail ("Couldn't load
Searcher: " + ex);
    }

    Query q1 =
searcher.makeQuery("word");
    Result r1 =
searcher.find(q1);
    assert(r1.getCount() == 1);

    Query q2 =
searcher.makeQuery("notThere");
    Result r2 =
searcher.find(q2);
    assert (r2.getCount() == 0);
}
```

SearcherFactory

Where does a Searcher come from? Currently, that's left up to whoever calls its constructor. Instead of letting clients

depend on the constructor, we'd like to introduce a factory method responsible for locating the Searcher. (For the test, we'll put a file "test.dat" in the directory for testing. If we wanted to be less lazy, we'd have the test create and delete the file as well.)

```
public void
testSearcherFactory() {
    try {
        Searcher s =
SearcherFactory.get("test.dat");
        assert (s != null);
    } catch (Exception ex) {
        fail ("SearcherFactory
can't load: " + ex);
    }
}
```

We can implement:

```
public class SearcherFactory {
    public static Searcher
get(String filename) throws
IOException {
        FileReader in = new
FileReader(filename);
        Searcher s = new
Searcher();
        s.load(in);
        return s;
    }
}
```

Now, a client obtains a Searcher by asking a SearcherFactory to give it one.

Looking Back

I'd like to put a design hat on, and look at the methods we've developed, from two perspectives: the search client and the Searcher class. Who uses each public method?

Search Client	Searcher class
Document.getAuthor()	new Document()
Document.getTitle()	Document.matches()
Document.getYear()	Query.getValue()
new Query()	new Result()
Result.getCount()	Result.add()
Result.getItem()	
Searcher.find()	

Looking at the Document and Query classes, I still have twinges that say they may not be doing enough (being not much more than a "data bag"). But both seem like good, meaningful "near-domain" classes, so we'll hold off on any impulse to change them. The Result and Searcher classes feel like they have the right balance.

What about the development process? It seemed to generate some blind alleys. For example, we had to change data structures from arrays to vectors (twice!). Is this a flaw in our process? No, it's not. The array was an adequate structure when it was introduced, and it was changed when necessary. *We don't mind blind alleys, as long as they're never one-way dead ends.* We're not omniscient, so there will be times we need to change our minds; the key is

making sure we never get stuck with a bad or over-complex design.

Moving Forward: Interfaces

The implementation we've derived above is a good starting point, but is not in final form. In real systems, the bibliographic information is often kept elsewhere, perhaps in a database, XML file, on another network, etc. We don't want our clients to know which alternative they're using.

The methods in the "Search Client" column of the table above show the interfaces required by clients. "Query" is probably OK as a class (since clients have to be able to construct them), but we would like to introduce interfaces for Searcher, Result, and Document. We'll apply the "Extract Interface" refactoring (from Fowler's book).

Unfortunately, the names we'd like for our interfaces are the same as the ones we already use for the classes. Since we'd like things to be better from the client point of view, and the classes so far are based on strings, we'll rename Searcher to StringSearcher, etc. and reserve the shorter names for the interfaces.

So, move Searcher.java to StringSearcher.java. Fix every call site and reference. Run the tests to verify that we've renamed correctly.

Introduce the interface:

```
public interface Searcher {
    public Result find(Query q);
}
```

(Run the tests.) Make StringSearcher implement the interface. (Run the tests.) Now, the only place that must reference StringSearcher by name is the SearcherFactory interface. (We could remove that dependency, and perhaps also put the String* objects in a different package, but we won't do that here for reasons of space.)

Apply the same process to Result, renaming the old Result to StringResult, and introducing the interface:

```
public interface Result {
    public Document getItem(int
i);
    public int getCount();
}
```

The StringSearcher class should still construct a StringResult object, but its return type should remain Result. (We don't mind if the String* classes depend on each other, but we don't want to make clients aware of that fact.)

Finally, introduce the interface for Document:

```
public interface Document {
    public String getAuthor();
    public String getTitle();
    public String getYear();
}
```

We're left with two concrete classes that clients will depend on: SearcherFactory and Query. Clients depend on the interfaces for Searcher, Result, and Document, but not directly on the classes implementing those interfaces.

Conclusion

We've developed the bibliographic system's model in a typical Extreme Programming style, emphasizing simple design and a process of alternately testing and coding. The unit tests supported us in designing, coding, and refactoring. The resulting system could have either a simple command line or a graphical user interface attached to it.

Resources and Related Articles

- This article is the basis for chapter 1 in *Extreme Programming Explored*, by Bill Wake.
- search.zip (preferred) or search.jar contains all Java code.
- "The Test/Code Cycle in XP: Part 2, GUI," William Wake. (A chapter in *Extreme Programming Installed*, by Ron Jeffries et al.)
- *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.
- *Refactoring: Improving the Design of Existing Code*, Martin Fowler, Addison-Wesley, 1999.
- JUnit home
- Test-First Challenge

Translations

- Japanese: Part 1, Part 2. Courtesy of Shinichi Omura.

[Written 1-25-2000; re-titled and revised 2-3-2000; added search.zip 7-2-00.]