XP In Smalltalk – Why it's Ideal

Smalltalk has been around for a long time now – nearly thirty years. In that time, the syntax of the language has been remarkably stable – the language started out with 5 reserved words and two operators – it has the same number of both things now. Contrast that with far newer languages, like Java and C# - every time someone comes up with a new feature that these languages should support, a new set of reserved words is born (see the 1.5 release of Java for a raft of examples of this).

One question to ask yourself is, "why is that?" Well, Smalltalk is built on a small number of fundamental assumptions:

- You don't know everything
- You can learn what you need to know by building incrementally
- The fundamental operating principle is *messaging*- objects interact by sending messages to each other

These fundamentals make Smalltalk a very powerful tool in general – and in particular, they make Smalltalk ideal for an XP (eXtreme Programming) project. Let's take the assumptions above in order.

1. **You don't know everything, but you can learn by building incrementally**

Smalltalk is ideal for exploratory programming. From incremental programming, workspaces, inspectors and debuggers we get a toolset that fully supports iterating towards the correct answer. Most Smalltalk developers start by opening up a workspace and writing a few lines of script. They then highlight the block of code and execute it. At this point, a trip to the debugger is very common, and this is a **good thing**. In Smalltalk, an unhandled exception **throws you into the debugger**. Additionally, the debugger is looking at the suspended system, not at what happened in the past. Not only can you inspect variable state and step through code – you can change the code and rewind the context stack. This makes it extremely easy (and inexpensive) to explore a possible route through the system. Instead of having to create a whole body of code before we can do a test, Smalltalk allows for *informal* testing before we even get to actual development. Never mind unit tests; a Smalltalker can do basic exploration before he even gets that far.

Here's an example from BottomFeeder, an open source RSS/Atom news aggregator written in VisualWorks Smalltalk:

```
doc2 := Constructor
        documentFromURL: 'http://www.cincomsmalltalk.com/rssBlog/rssBlogView.xml'
        forceUpdate: true
        useMaskedAgent: false.
cls2 := Constructor determineClassToHandle: doc2 content.
target2 := cls2 objectForData.
feed2 := cls2
        processDocument: doc2 content
        from: ' http://www.cincomsmalltalk.com/rssBlog/rssBlogView.xml'
        into: target2.
```

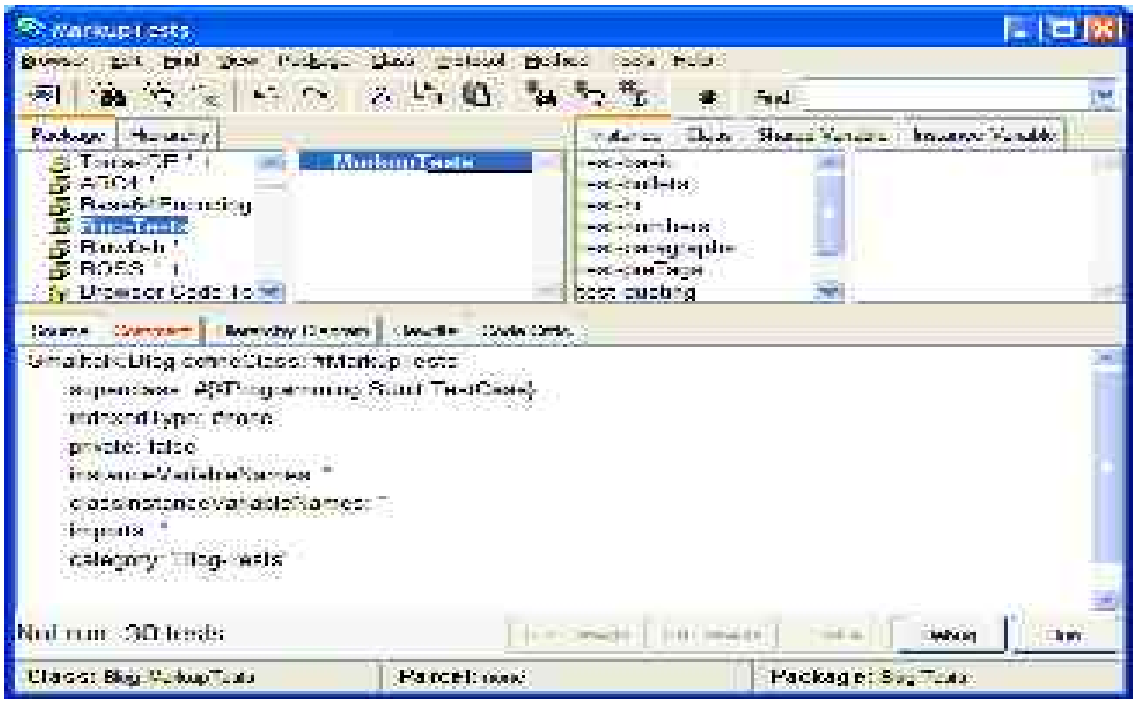
That code is a condensation of how BottomFeeder grabs content from the web, parses it into an XML document, and then decodes that document into an RSS feed. Now, I didn't stumble into that all at once. I started with something like this:

```
| xml xmlDoc |
xml := HttpClient new get: 'http://www.cincomsmalltalk.com/rssBlog/rssBlogView.xml'.
xmlDoc := XMLParser on: xml readStream.
```

The basic code in BottomFeeder started out looking far more like that, using the basic VisualWorks libraries. By exploring things in a workspace, I was able to *iterate* towards more functional code – and then I was able to *refactor* the code by further workspace exploration. Smalltalk supports exploratory programming via the interactive tools and that in turn supports the basic XP tenet of YAGNI (You aren't going to need it). You can create working code quickly, and translate that from simple workspace scripts into a basic set of classes very quickly. As actual requirements are discovered, you can continue to iterate towards a better solution.

So incremental development, with full support from the environment is a big help. However, XP advocates test-first development, and all I've talked about is a very informal form of test first. Well, as it happens, test-first works very well in Smalltalk. This isn't a huge surprise, since it **was invented there**. SUnit is the grandfather of things like JUnit, NUnit (etc.).

Once you have a handle on what you want to do (after some basic workspace exploration), you can proceed to writing tests – and to following that with code. I'll use an example from the Blog posting tool that ships with BottomFeeder. That tool supports some basic "wiki-style" markup. I created that support via test-first development – I knew what the HTML that came out of the markup converter was supposed to look like, and I knew what the incoming markup was supposed to look like. First, have a look at a VisualWorks browser with test support:



```
Smalltalk.Blog defineClass: #MarkupTests
    superclass: #{XProgramming.SUnit.TestCase}
    indexedType: #none
    private: false
    instanceVariableNames: ''
    classInstanceVariableNames: ''
    imports: ''
    category: 'Blog-Tests'
```

Not run: 30 tests

Class: Blog.MarkupTests   Parcel: none   Package: BlogTests

Do you notice the "Run" and "Debug" buttons at the bottom? I've selected a package with 30 tests – I can either run them all, or go down to any level and run just a few (or one) test. If I can't figure out **how** it's failing, I can debug it – step through the test (which in turn, steps through the code) and see what's going on. As I figure it out, I can fix the code **in the debugger or in the browser** – depending on how it falls out in testing. Let's drop down to a specific test:

```
testBulletsWithMarkup
	| result start |
	result := 'This is a test of bullet markup
<ul>
<li>Line 1 <b>with bold text</b></li>
<li>Line 2 <i>with italic text</i></li>
<li>Line 3 <u>with underlined text</u></li>
</ul>
That should do it'.
	start := 'This is a test of bullet markup
*Line 1 !with bold text!
*Line 2 ^with italic text^
*Line 3 _with underlined text_

That should do it'.
	self should: ((MarkupHelper from: start readStream doAllMarkup: true) = result)
```

The code shows the (presumably) resulting HTML, along with the incoming markup. Running the test should result in a green bar at the bottom of the browser – if it doesn't, the debug button is right there, ready to show me what's wrong.

This makes for an ideal test-first scenario. First, I write the test. It fails, since the code isn't there yet. Then I start writing the code, re-running the test as I make progress. I can create methods from inside the debugger – if it spots a method that doesn't exist, it will allow me to create it. From test-first I can jump straight into the kind of iterative development I spoke of above. Instead of having to walk through lots of steps:

- Write test, compile
- Write code, compile
- Run test, which fails
- Return to top

I can instead stay at step three – as I'm running the test, I can iteratively work on the solution (or terminate it and go straight into my code browser if that's what I want to do). Smalltalk tools offer a completely iterative experience in test writing and execution – you can keep yourself completely focused on the problem at hand instead of having to jump in and out.

2. The fundamental operating principle is messaging between objects

This goes back to the reserved word comments I made in the opening paragraphs. Smalltalk is **simple** – 5 reserved words, 2 operators. New developers can learn the syntax in minutes, and then start in on the important part – the class libraries (and with that, the importance of inter-object messaging). This plays right into the ability to just jump in and start exploring – something which is crucial to success with XP.

If you can't jump in and quickly create tests – either the informal sort I mentioned at the beginning, or the more formal sort I spoke of in the last section – then you'll end up being bogged down by **trivia**. Instead of haggling over the appropriate syntax, Smalltalk developers are immediately working with objects and interfaces. Instead of deploying squads of language lawyers, they end up working requirements. Since XP is completely focused on working for (and discovering) the requirements, Smalltalk is ideal – it removes the artificial barriers that many other systems toss into the mix. Ultimately, the basic Smalltalk strengths – simplicity, testing support, and development tools – enable a perfect storm of support for XP.