Turning Applications into Web services via their User Interfaces

MAT's Ephphphatha Studio gives API (Application Program Interface) to any applications with graphical user interface at their highest level of abstraction; it is the fastest way of turning an application into a Web service.

Much effort has been made to make intellectual property, captured within applications, available to business partners and customers, and has given birth to a myriad of technologies, including Web services. This effort, however, has conveniently assumed that these applications already have ways for other applications to talk with them. This assumption is untrue in many cases; one only needs to look at typical mainframe applications. Many of these incommunicable applications, however, still interact with human users via their user interfaces, and *Ephphphatha* takes advantage of this to make them communicable; by pretending to be a human user on behalf of other applications, *Ephphphatha* converts an incommunicable application communicable, all without ever touching the application implementation.

Key technologies in *Ephphphatha*; windows messaging and shadow components

Windows applications interact with their human users via keyboard strokes and mouse movements/clicks; with a keyboard and a mouse, a human user types letters or letter combinations in edit fields and clicks on buttons to give proper stimuli to the applications to indicate what it is that he/she desires from the applications.[1] The applications, then, respond to the stimuli with a certain change on the screen, such as typed letters, changed colors, different pages, or newly populated fields, etc. The very Win32 technology that makes this interaction possible is based on Windows messaging; a stimulated Windows component sends out the information regarding itself and the stimulus, and the other components that are interested in the stimulus on this particular component receive and process the information. *Ephphphatha* is built on this technology.

The Windows components basically do two things; accept user stimuli and relay the applications' responses to the stimuli back to the user on the screen. Although they do much more than these two things inside themselves, when observed from outside, these two things are the only behaviors they perform. In distributed computing, there is a concept of an interface, which defines the behaviors of a component while hiding its implementations. A component that works on behalf of another component by shadowing it from outsiders is called a shadow component. When an outside component gives a stimulus to the shadow component, the shadow component relays the stimulus to its shadowed component it represents, and again relays the response from the shadowed component back to the outside component. This technique is used

[1] Note that edit fields and buttons are loosely defined in this context. Edit fields are used to indicate any area on the screen where a user can type, and buttons are any click-able areas on the screen. Collectively, they will be referred to Windows components, or components, throughout this document.

in order to leave existing applications untouched while being connected to other applications.

How *Ephphphatha* mimics Human Interactions

Ephphphatha enables developers to create a shadow component for any Windows component. With shadow components, along with Windows messaging technology, a developer can monitor any keyboard or mouse actions performed against an application. If a user presses a key on a keyboard in an edit box field, the edit box will send out information regarding this event, including its corresponding shadow component. *Ephphphatha* can be programmed to take any actions with this information. Furthermore, this shadow component can also initiate a stimulus to its shadowed component. It can basically tell the component that a key has been pressed as if it is a human user, and the shadowed component won't know the difference.

An example can illustrate this further. Let's take MS Outlook. In order to send an email to someone, one would press the "New" button, and a new Windows for the email would pop up. Then, while in the email Windows, one would type an email address in the "To" field, type in the subject and contents in its appropriate fields, and hit the "Send" button. This series of actions can be exactly duplicated with *Ephphphatha Studio*. A developer can "scan" the graphic user interface (GUI) of Outlook and convert the GUI components such as "New" and "Send" buttons as well as "Subject" and "Content" fields into programmable objects or shadow components in the Studio. Then he/she can now tell these shadow components about the events and associated information such as "New" button has been clicked, "To" field needs to be typed with an email, "Content" field is to be filled with whatever message, etc. in the codes. MS Outlook will behave exactly the same way.

Ephphphatha architecture in Web services

Ephphatha consists of two parts; generating objects that represent the components and invoking these objects. Generating objects is essentially drawing a map with which to locate window components of interest. More specifically, it is 1) selecting the window components of interest, 2) persisting information with which to locate the selected components at a later time, 3) providing programmatic control over the default behaviors of the components as encapsulated objects, 4) combining #2 and #3 to generate objects that programmatically represent the user interface components. Ephphatha Studio is a desktop application that automatically provides #2, #3, and #4 – all that a user needs to do is select the window components he/she wishes to programmatically control.

Once the map has been generated, windows messages can be sent to the components on the map. Ephphatha hides the complexity of windows messages in its API; for example, button.click() internally generates a windows message that simulates a mouse click and sends this message to the targeted button component.

A word on using native APIs vs. APIs made by *Ephphphatha*

The following list of issues should be examined closely when deciding whether to use native API of an application or that provided by *Ephphphatha*.

- Transaction volume – if the web service needs to scale to handle a large volume of transaction, going through user interface may not be fast enough
- Development time – user interaction is the highest level of abstraction possible in any application. What this means in development is that a significantly less number of APIs need to be invoked to do the same thing. This not only means less coding, but also implies easier debugging
- Complexity of business flows – with *Ephphphatha*, a business process is defined by screen shots of applications, not in words. When dealing with complex business processes, screen shots usually avoid misunderstanding in requirements documents.