NPL for the Net: Summary of Possible Approaches

by Alan Green

January 20, 1999

The purpose of this document is to recap the possible approaches to building "NPL for the Net" as gleaned from discussions with some of the community's leading developers. At this point I'm organizing what we've learned, but not yet taking a position on it.

Next I'll send it back around to those who contributed ideas so that they can see and comment on what others have proposed. We'll make a decision matrix to evaluate each idea. After another round of discussions, I'll recommend a course of action to Niakwa and the community.

Late last month, we contacted many of our most respected NPL developers. (We will include other important developers in this process who we weren't able to reach in our first round of calls.) We asked them how we can help the NPL community to:

1. Connect their NPL applications to the Internet (or intranet, or network, or operate standalone) using a browser interface.

2. GUI-ize and Net-enable their entire (potentially huge) application including browser interface and transaction processing) in under a week.

3. Deliver this solution in 1999.

Input was solicited and received from:

Donovan Burkhart
Chris Cummings
Russ Fairchild
Craig Freeman
Pat Legg
Mike Liston
Chris Maxwell
Sam Matzen
Tim VeArd

Niakwa, Inc., 1850 West Winchester Road, Libertyville, IL 60048 Phone 847-816-7400 Fax 847-816-7420

Possible approaches identified so far boil down to:

Low Developer Effort/Code Impact Solutions:

Provide a Screen-Scraper

Support an NPL Telnet or Terminal Emulator Session in a Browser

Medium Developer Effort/Code Impact Solutions:

Add HTML Form Definitions to Existing NPL Code

High Developer Effort/Code Impact Solutions:

Make Existing NPL Apps Event-Driven, and Vinny-ize Them

Automated Tools:

Auto-Generate HTML and Vinny Screens with a Code Parser

Provide an Automated Event-Driven Conversion Tool

Workbench Add-Ins:

Provide a Workbench-Based Event-Driven/GUI-izing Conversion Tool

Support Workbench-Assisted Application Re-deployment

Transform NPL:

Make NPL a Java-Like Language

Make NPL an Object-Oriented Environment

Middleware Layer:

Provide Middleware that GUI-izes and Net-Enables

In addition to the phone discussions, several participants also sent written follow-ups. In the discussion of NPL as an object-oriented environment, I have also included notes and correspondence from a discussion held by several of us in December 1997.

Preliminary Observations

(**Pat Legg**, phone discussion) First we need to define what we're trying to accomplish and state what it will look like when we're done:

> Allow legacy apps to operate as clients on the Internet as well as on local nets.
>
> GUI-ize legacy applications.

The problem is that browsers are not logical front-ends for legacy apps.

(**Chris Cummings**, e-mail message) "Don [ovan Burkhart] and I got real familiar with this during our quasi-GUI research project a couple years ago <grin>. We solved it for SPEED by "simulating" an event-driven app. We added capability here and there, but we could not force the software to do something it could not ordinarily do. For instance, to handle the mouse, we took over keyboard input and put in a keystroke queue. If the user clicked on something, we would figure out what keystroke equivalents would be required to do "that" or go "there" and feed them in as if the user had actually typed them in.

For instance, if the user was editing field #2 in a file maintenance situation and clicked on field #6, we might feed in a C/R to finish off field #2, let SPEED and the app check it for errors, and if none, feed in the keystrokes to get the "focus" to #6. With all the editing modes and phases in file maintenance, interactions with programmer-defined special keys, and the SPEED event callbacks where the app could move the user around on its own, it got hairy keeping track of where we were, what to highlight, and how to get to where the user wanted to go, all the while remaining 100% transparent. We couldn't have done it without the underlying framework of SPEED to enforce -- or at least encourage -- app programmers to use standardized methods for common tasks and allow for certain "events" in their programs.

After what I've seen recently at [some large customer accounts], there are some truly awful, barely supportable, procedural quagmires out there that still get the job done. Managing event-driven navigation requires a degree of organization, just to keep track of valid events, that is inconceivable at these two locations without a rewrite. And I suspect the issue isn't just "restructuring" an app to make it GUI-able. You also have to restructure the support layers, and maybe even the author's business. If I rewrote somebody's app, GUI-ized it, and handed it back, could they still support it, or would they be up a creek without a paddle?

As I see it, "GUI" means an NPL app gets complicated in three important ways:

> First, are the "events" and "objects". Certain chunks of code must be designed to execute in any order at any time. Keeping things straight requires pre-planning,

period. SPEED actually HAD objects in a real sense (files, screens, and fields) and a handful of standardized events. Without a middleware layer like SPEED designed for this kind of interaction, our efforts would have been more than "daunting". We occasionally ran into someone who'd made a minor change to SPEED or did something else in a non-standard way, and SideFX was dead in the water. With few exceptions, we required absolute adherence to the SPEED paradigm.

Second is "functional expectations." You don't have to support every common GUI or browser capability, such as MDI and frames, but there is a **minimum** set, such as moving the caret with the mouse, cut & paste, drop-down menus, etc. that just about everybody expects from a GUI app. It may not be easy to retrofit existing apps to even these minimums, much less expect that the latent capabilities are already there.

Third is "aesthetic expectations." GUI apps have a different look. This is an important lesson from SideFX. GUI screens are made from a different toolbox than we have in NPL, and the translation of one to the other is awkward. The same skills we've honed in making pretty character screens don't make us into commercial artists and graphics designers either. "Presentation" is (unfortunately) pretty important, and I doubt we can add enough smarts to a graphical translation algorithm to compensate for the human senses of balance, beauty, and effect. We had ideal conditions with SPEED in that we were dealing with predictable, data-driven objects with identifiable properties. And we still blew it. Some of it was undoubtedly related to our dubious graphical design talents, but a significant portion is probably due to what we had to work with.

(**Mike Liston**, e-mail message) There are three stages in terms of their impact on existing NPL code and the effort to transform it:

1. **No (or low) impact/effort.** Example: A program which auto-converts legacy spaghetti code to event-driven.

2. **Moderate impact/effort.** Examples: A solution auto-gen'ed by a screen-scraper and Mike's "middle way".

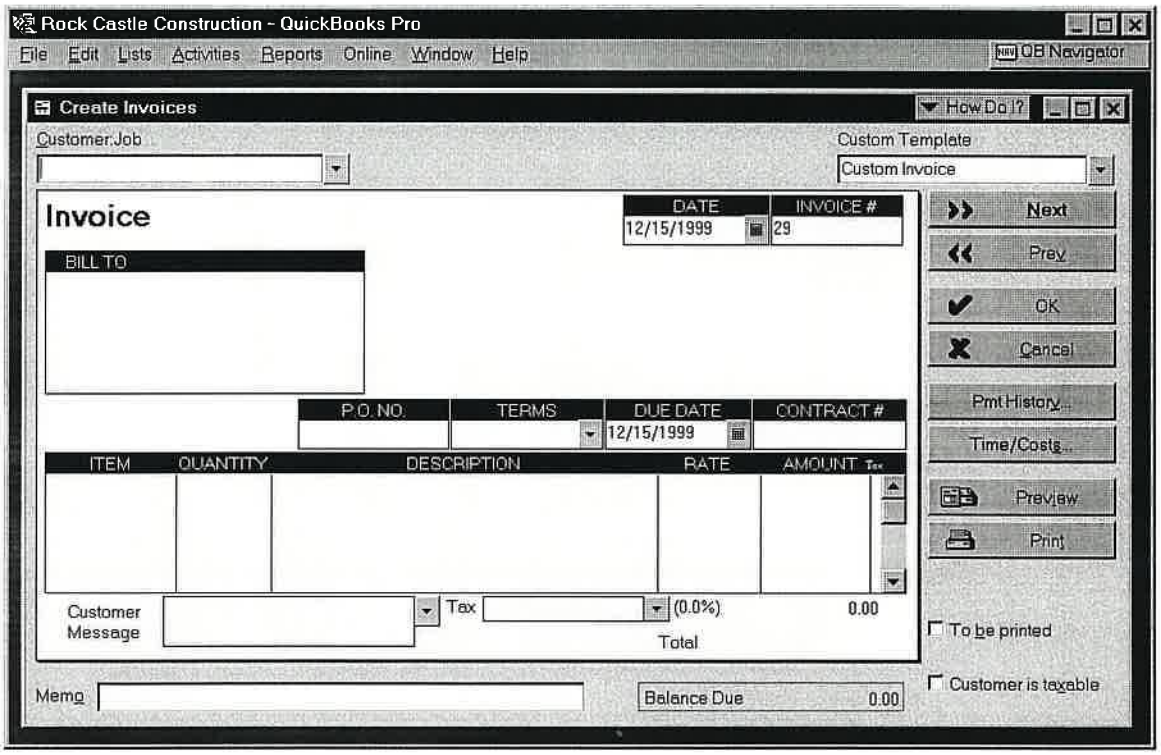
3. **Heavy impact/effort.** Example: Full restructuring apps to be event-driven and (manual) Vinny-ization.

(**Tim VeArd**, phone discussion) Open NDM is more important than GUI-izing.

(**Sam Matzen**, document "Decision 1999") Modern User Interface Presentations. There are two entirely different user interface presentations deployed in the modern environment and three sub-types that must be considered:

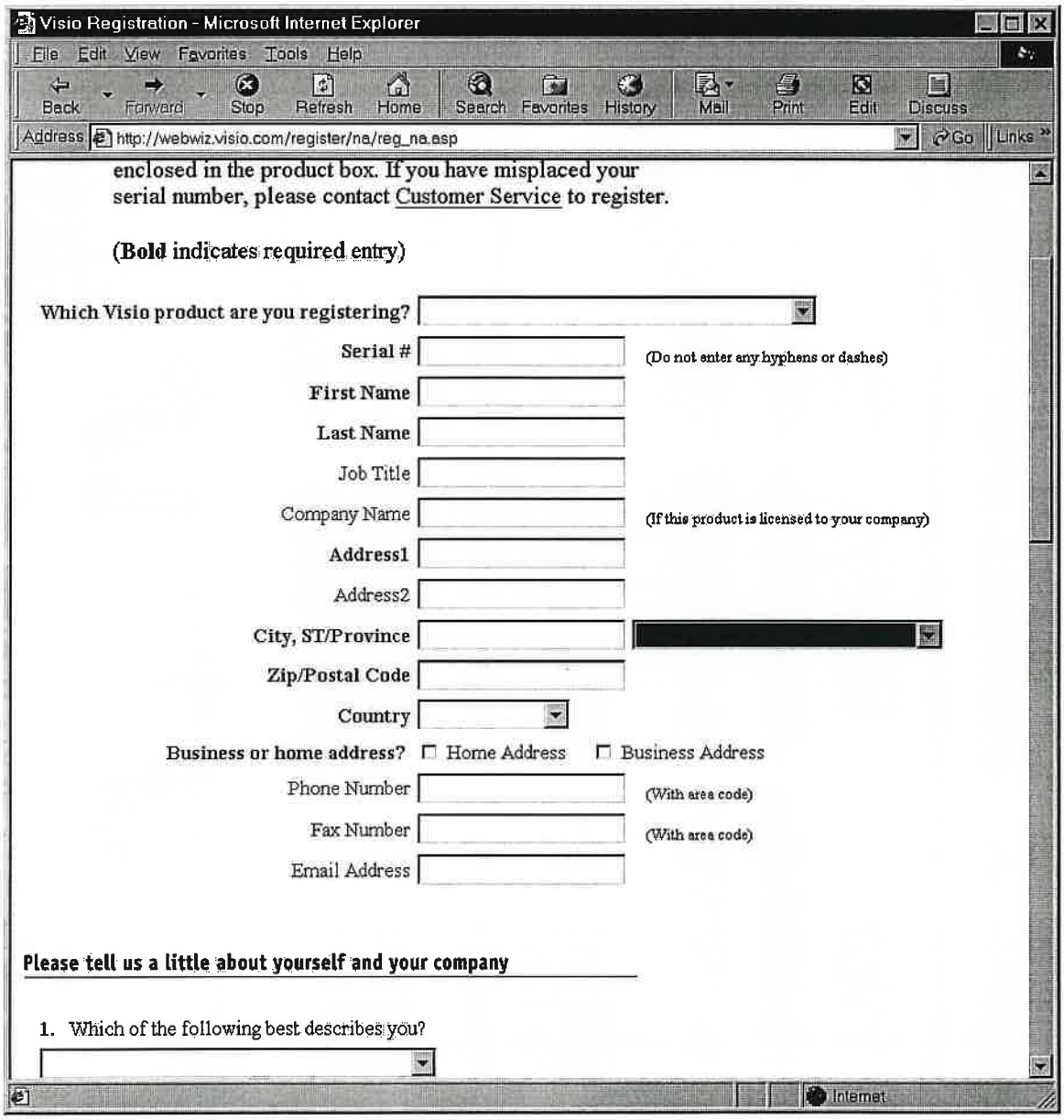
Windows. Form based input with almost unlimited display control options including combo boxes, radio buttons, control buttons, labels and text boxes, and many kinds of grids. Flow control is handled with drop-down menus and button bars. There is typically a main program that contains multiple forms (documents) called a multi-document-interface, thus multiple forms can be active at one time. Within the Windows interfaces there are three different end user experience levels to consider

> **Consumer** – Home users and consumers with limited computer experience. Microsoft Money is a good example of a consumer product.
>
> **User** – Experienced computer user. These are users, as we now know them.
>
> **Managers** – Inexperienced users that require special interfaces for the presentation of summary management data with links details presented in a management reporting style.



Web. Form based input with limited display control options. Most web applications provide not-only form input, but informational content. A Typical e-commerce site provides the user primarily with information content and uses form based input to extract information from the user. The flow of a **Web** application is vastly different than the flow of a **Windows** application. The application must

be designed to work on different web browsers and widely varying display geometry.



Note: A typical application provides a complete **Windows** interface and a limited number of **Web** forms.

User Interfaces. Different applications require different user interfaces. Currently these interfaces need to be considered:

Microsoft Windows on a workstation (pc)
Microsoft Windows on a Windows Terminal
HTML based web applications

Windows Terminal launched by a web browser
Character terminal
Character terminal emulator on a workstation
Graphical interface emulation of character based application
X Windows on Unix
Mac Windows on Macintosh

Modern User Interface Behavior. Modern application user interfaces exhibit a number of behaviors that are foreign to legacy BASIC2/NPL applications:

Menu structure and program flow control are an integral part of the user interface (Windows). Most legacy applications have some form of menu system for program selection, with additional program flow control implemented in the business logic.

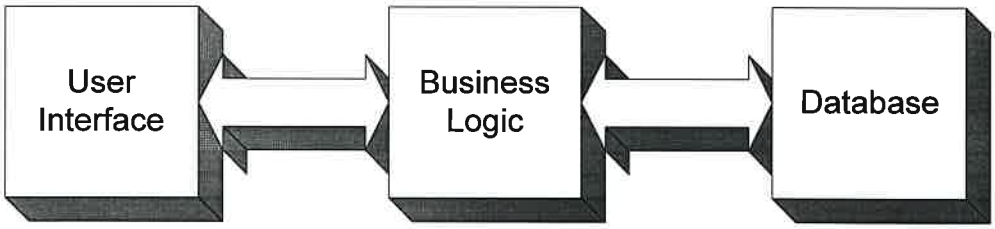
Event processing. The windows environment requires that the user be able to move the cursor from one input field (text box) to another with the mouse. Many legacy applications have program logic that leads the user from one field to another.

Transaction processing. The web interface is generally a transaction processing style interface where the user enters data into a form and submits the form for processing. Most legacy applications process data field-by-field.

Reporting. Most reporting requires some form of user interface for report selection entry and subsequently the report is prepared and printed. Separating the user interface logic from the report logic can be difficult with some legacy applications.

The Whole Picture. I believe that we need to study the whole picture before we start trying to solve the problem. In other words, let's take a detailed view of the business application components required to deploy in a modern environment:

User Interface
Business Logic
Database



User Interface Component. There are typically two primary user interface classifications:

Display. In the display interface, information is displayed on some form of display and the operator responds interactively with the display with a keyboard, pointing device, voice, or other human/computer interface.

There are three broad categories of display interfaces into which each of the currently used interfaces can be placed:

Character

Character terminal
Character terminal emulator on a workstation
Graphical interface emulation of character based application

Windows

Microsoft Windows on a workstation (a workstation is a pc)
Microsoft Windows on a Windows Terminal
Windows Terminal launched by a web browser
X Windows on Unix
Mac Windows on Macintosh

Web

HTML based web applications

Report. The report interface allows the user to select what is to be printed, and the print details are processed through a printer driver and subsequently a printed document is prepared.

There are two categories of report interface:

Character. These are reports printed on character printers. Generally they are of a single type style, character width and lines per page

Graphical. These are reports that include variable type styles, lines, graphics and pictures.

Business Logic Component. The business component includes all program logic used to provide a bridge between the **user interface** and the **database**.

Database Component. The database component includes the storage and retrieval of persistent data.

Low Developer Effort/Code Impact Solutions

Provide a Screen-Scraper

(**Craig Freeman**, phone discussion) Spray a 3D background behind existing text. Look at jsb.com for a product which takes text-mode UNIX apps, parses their generated screens, and GUI-izes on the fly by providing a 3D background to the text.

(**Craig Freeman**, e-mail message). The concept works by automating the object creation process and then disabling all the objects so that the mouse can't accidentally break the procedural flow of an existing NPL application. But when the program requests data entry at a specific point on the display, NPL determines if an object exists at the position and if so, enables the object and starts behaving like a true Windows app. Once the program exits the control, it is once again disabled and a value is returned.

As a start, text areas could be automatically modified so that a 3D text box was drawn around field input areas according to one simple rule: If the current NPL "color" matches that established for an "input field", PRINT AT commands followed by a string would create a text box with a height equal to 1/25th of the NPL window height and a width equal to the greater of the string length or the third parameter of the PRINT AT, e.g. PRINT AT(x,x,thisvalue). This would handle the 90+% of all displays that spray labels on a background and then position field values in a second color (or bright, etc.)

Now that the displays look and feel like Windows apps, and data entry would need a little help. Since every NPL developer already has some kind of KEYIN routine we need only supply a set of replacements for text, numeric and date entry that work like Windows functions but optionally call a validation function with every keystroke to let the coder handle special events.

That leaves Buttons, List Boxes, Combos, Images and other objects that usually do not fall within a procedural flow. For this class of objects we enhance NPL's syntax to display a control (disabled). The sample program below would display a list of all clients and a button for exiting the program.

```
; Create a labelled button
PRINT $BUTTON AT(Row_10,Col_20,Width_6);" Exit"
;
; List every customer
SET LISTBOX.Forecolor = _nplRed
PRINT $LISTBOX AT(12,20,ListBoxRows,ListBoxWidth,_nplSorted)
        CustArray$()
;
SET FOCUS AT(10,20)
```

```
END

DEFFN' NPL_LostFocus(Row_Col,Value,Value$)
SWITCH Row_Col
        CASE 1020
                ButtonState=Value
                RETURN CLEAR
                GOTO EOJ
        CASE 1220
                CustomerName$=Value$
                RETURN CLEAR
                GOTO FieldAfterCustomer
END SWITCH
RETURN
```

When FOCUS is set to a given screen position, NPL activates the control located there. When focus is lost, NPL would return the default value for the control (usually .Text$ or True/False) and would call DEFFN' NPL_LostFocus passing the row/column value of the upper-left corner of the control.

(**Sam Matzen**, phone discussion) For a similar tool, look at CA's Opal.

(**Alan Green**, comment) GUIdance also does essentially the same thing.

(**Chris Maxwell**, phone discussion) Use a screen interpreter such as his to parse NPL screens and gen VB screens on the fly. Note that this is different than a code-parsing solution. Instead, you're analyzing and reproducing in GUI the actual output screen. His version is much better than Instant Vinny but not as good as a designed screen.

This solution can be keystroke oriented (passes back each keystroke) or field oriented (passes back the entire field content)

(**Russ Fairchild**, phone discussion) He is very pessimistic about screen interpreters, because their output is ugly.

Support an NPL Telnet or Terminal Emulator Session in a Browser

(**Mike Liston**, e-mail message) This fits in his "Low impact/effort" stage. Characteristics are:

Character-based or browser-based.

App look and feel are unchanged.

Client interacts with the server in the same manner as a dumb terminal interacts with a minicomputer.

Like turning a web browser into a Wang 2236 terminal!

Little or no changes required to existing NPL applications.

Might be nice if LINPUT could be handled as a textbox to look a little more GUI.

Helpful to have a TrueType NPL font with the IBM PC symbol set.

Requires low latency so is currently practical only on intranets, not on the internet.

Possibly extend this idea with these enhancements:

When NPL encounters a LINPUT, create a textbox.

For KEYINs we should supply a "make a textbox" verb.

Change $BOXTABLE to permit:

Character boxes
True boxes
3D boxes under Windows

(**Sam Matzen**, document "Decision 1999") Support a server-side NPL app hosting a thin client (a terminal emulator such as Soft60) in a web-browser. A client session could display labels and text boxes. Problem to overcome: sessions are stateless (user can push Back button), so app must be able to handle incomplete transactions.

Simply stated, it maintains the existing application with little or no modifications and puts a more modern presentation on the fields. By using the 16 attributes, an interface emulator will capture the labels and text boxes and display them on the windows screen in standard windows format. The developer/user will be allowed to modify the presentation styles and colors, but the application will not change. The menus, data entry screens, reporting and database will not change.

For the **Unix** based applications using dumb terminals or terminal emulators on personal workstations, deployment of a graphical interface emulator will be quite easy. A web browser can launch this interface emulator, thus the application can be immediately web enabled.

For the **Windows** based applications, we will probably need to provide a tight interface between the graphical interface emulation and the windows NPL runtime, a relatively simple task since the runtime already supports remote control. At this time the only way I know of getting these applications web-enabled is to put them on Unix where we launch the application from a web browser. There may be a way to make this happen on NT, but I am not familiar with it.

Up Side:

Can be deployed quickly.

The existing application will need minimal to no changes.

Down Side:

This approach has been implemented by other companies and has met with limited success.

Latency may be an issue for those applications deployed on the web.

The graphical interface emulator leaves all of the logic with the existing application.

Since this solution doesn't modify the application logic (which includes the user interface), each character must be sent to the application, processed and returned to the user interface, thus latency.

Typical latency with a 56k connection will be in the 100ms range. For some applications this is acceptable, for some it isn't.

This will not provide a web page-style block-mode interface.

I believe that we can provide some relief to the community by providing the **Graphical Interface Emulation** solution. It isn't for everyone, but may provide some relief for those VARs that are under a lot of pressure and aren't sure what to do.

Can we satisfy the stated objective of connecting their NPL applications to the Internet using a browser interface? This is highly application dependent, and will depend on what types of data are to be extracted from the web interface. For most applications it is not practical (or desirable) to make all components applicable to the web. However, all components can be deployed in Windows and a few of the consumer class forms implemented on the web.

Medium Developer Effort/Code Impact Solutions

Add HTML Form Definitions to Existing NPL Code

(**Mike Liston**, e-mail message) This is Mike's "middle way" (moderate impact/effort). The advantage is that you don't have to restructure the code to be event-driven. Characteristics:

Form-based.

Browser-based.

Stateless.

Target application "look and feel": Web pages.

The NPL app provides information for the creation of a HTML form which is transmitted to the web browser at the client.

The end user enters all data on the screen.

Performs as much data validation as possible without interacting with the server.

The NPL app only needs to respond to "submit" and "back" from the client.

The user-interface portion of existing NPL applications must be modified, but the underlying business logic can remain substantially the same.

Existing procedural code is usable because the entry and exit points are predictable.

The more limited set of controls available in a web browse may make for a shorter learning curve than the entire gamut of ActiveX controls.

This approach is practical for use over the internet as well as on intranets.

We should supply a small number of controls, such as:

Textbox
Command button
Scroll bar
Something to print graphical stuff as needed

Other characteristics:

This helps the developer by reducing the number of controls he has to deal with.

The user can operate on any field on the form.

Provide limited data validation.

Enable pop-ups, drop-downs.

This will look and operate as if it's event-driven. But, this approach will still have to send batches of data input responses back to the NPL program on the server.

"An app doesn't have to be truly event-driven to look event-driven to the user."

High Developer Effort/Code Impact Solutions

Make Existing NPL Apps Event-Driven, and Vinny-ize Them

(**Mike Liston**, e-mail message) This is Mike's "heavy impact/effort" stage. Characteristics:

Target application "look and feel": MS Outlook.

Conceptually like putting the VB side of Vinny at the client and communicating over the network instead of within the PC.

All VB components may be used at the client.

Requires modifying NPL apps to be totally encapsulated so they can handle events in an unpredictable order.

May be practical only on intranets.

Automated Tools

Auto-Generate HTML and Vinny Screens with a Code Parser

(**Sam Matzen**, phone discussion) Auto-parse the PRINT and INPUT class statements in legacy code and generate Vinny (VB and HTML) user forms.

You will need late binding (set some parameters very late), and developer customizability including being able to rearrange the target display screen at design time. You must allow mouse click and edit on any displayed field (or a way to disable the field).

Provide an Automated Event-Driven Conversion Tool

(**Donovan Burkhart**, phone discussion) You can't GUI-ize without recoding.

Build an interceptor/redirector version of NPL, which would intercept certain statements (such as DATALOAD, LINPUT) and redirect them to commands in a given environment (SQL, Windows).

(**Mike Liston**, phone discussion) Provide a program that converts spaghetti code to structured. He says:

> For every 10 hours of original effort to write a program, it takes one hour to take it to structured form using functions and procedures.
>
> He's able to convert about 50 sectors of code a day, which is made easier because of his Toolkit.
>
> The conversion process is mechanical. He can watch TV and work on his laptop while doing it.

From prior discussions, Pat Legg believes that this process is not mechanical because there are too many variations in coding style. Even Mike says he doesn't know if this idea is doable without a large effort.

(**Pat Legg**, phone discussion) Regarding the code-updater idea (where an external program analyzes and restructures legacy code), we won't be able to advance much beyond Craig Freeman's MRCLEAN or Mike Liston's Toolkit because artificial intelligence analysis of NPL code is beyond the reach of science.

Workbench Add-Ins

Provide a Workbench-Based Event-Driven/GUI-izing Conversion Tool

(**Craig Freeman**, phone discussion and follow-up messages) Make legacy code event driven:

1. Clean up the code first (de-spaghetti-ize with MrClean).

2. Then take one long day to change 100 data input statements with lots of validation.

3. Use the Workbench to add RETURNs after each input, then add one big ON... or CASE... to call.

KCML-ize NPL: embed GUI management statements in legacy NPL apps. This would NOT require a fancy visual editor ... just some syntax enhancements to NPL.

In the crudest sense, the NPL window would be a single, resizable "parent" form. All other controls would be children. There would be no need to support the creation of additional forms that appear outside the borders of the main window (this really complicates things!) PRINT and PRINT AT commands to device /005 would create "labels" on the form. PRINT HEX(03) would destroy all labels and other controls. Note that 32-bits makes this possible because we can have as many controls at a time as we need.

Resizing the main window would resize all displayed fonts as well as controls. I would do this much like Word where you can only change Text Boxes in increments ... guaranteeing that you don't end up with un-displayable font sizes, blank areas on the right or bottom, etc.

I would recommend supporting the following "standard controls" (you might need to require developers to license VB Pro so as to gain legal access to the needed controls):

3-D Text Box
Message Box
Scrollable .RTF Text Box (for help,warnings, license agreements, etc.)
Button
List Box
Combo Box
Image Box
3D Frame
Check Box

Multiple check boxes within a frame
Spinner
3D Icon Toolbar (return value = which icon was clicked)
Menu (at top of window)
Help

When implementing the above I would shoot for the lowest common denominator and try to maintain VB syntax for marketing reasons. I would support only the most popular properties directly. In cases where other properties were needed, the developer would have to treat the control as a "third-party control" as described below.

Third-party ActiveX controls would be supported through a special $DECLARE sequence that would create a control array and provide optional default values:

```
$DECLARE ControlName$(VendorControlName$,
        .Type="Button",
        .FontName$="Arial"
        .Width=10
        .Forecolor=_nplBlack,
        .ReturnValue=_nplTrue)
```

Allow programmers to optionally give names to controls at the time of creation that NPL would maintain in a "control name stack" until a PRINT HEX(03) command is executed:

```
ControlName$="Exit_Button"
; Create the control
PRINT <ControlName$> AT(Row,Col,Width);Text$
; Set focus on a named control
SET FOCUS AT <ControlName$>
; This would also still work
SET FOCUS AT(Row,Col)
```

Allow WITH statement to set properties of both standard and third party controls:

```
WITH ControlName$
        .ForeColor=_nplBlack
        .FontName = "Arial"
END WITH
```

Some additional control commands:

```
Controlname$.Visible=_True
Controlname$.Enabled=_True
ControlName$.Move=Row,Col : ; Updates the NPL "control stack" and
repositions control
```

MainWindow.Height = hTwips :; Resize the NPL main window

We would need a way to spawn additional instances of NPL that could open at specific screen locations without wasting time checking security. This would produce the equivalent of loading a new form:

$SHELL NPL (RowTwips, ColumnTwips, Height, Width, NPLProgramToRun$, OptionalParms$)

It might also make some sense to provide a "Fast Load" option that would bypass most syntax checking when programs were loaded in a new window (you had to check the startup programs when you loaded the first window so the odds are 99.999999999% that they are undamaged).

If you want a visual editor, I suggest that you have developers use the one in VB5/6. Let developers design any form that they want and then save it as ASCII text. An NPL routine could parse the file and create a series of NPL program statements. Not only would it simplify new development, but it would be very helpful for resellers who started to design things in Vinny but never finished!

From what little I know of KCML's GUI features, my suggestions are at least partially in line with their methodology.

Also disable generated objects so that mouse clicks by a user on a non-current field won't break a procedural legacy app.

(**Sam Matzen**, document "Decision 1999") This approach would have the NPL Workbench parse an existing application in both static and active (while it is being run) modes and modify the existing application code to be more modern. This would require that the workbench be able to convert the existing procedural field-by-field applications into form-based applications which allow the user to enter all data on a form and then submit the form. Field relationship editing is limited and many of the display elements would need to be modified to work with the modern environment.

Program logic would have to be modified to work with the form paradigm, and for many of the existing legacy applications, these modifications would have to be extensive. A graphical workbench tool would need to be available to modify the display layout. The existing display real estate of 80x24 or 132x24 works well for character interfaces, but will need to be reworked for the graphical interface, especially the web interface where the user can change the display geometry.

Up Side:

Existing report programs would not require much modification.

The user interface would be extracted into a form and the report would remain essentially the same.

Much of the existing application logic can be preserved and the database won't have to be touched.

The new application will be able to coexist with the existing character based application.

Down Side:

The workbench tools can't do it all.

The developer will have to write some business logic to supply different display controls (combo boxes, radio buttons, ...) with the necessary data tables and values.

The most-used programs (generally 3 to 5 in each application) will probably have to be completely re-constructed to work with the form paradigm.

The menus and flow control will either have to be re-written or provide a non-standard presentation in the windows environment.

The presentation won't be full windows compliant.

After all this work, the application will probably have to be re-deployed anyway.

Support Workbench-Assisted Application Re-deployment

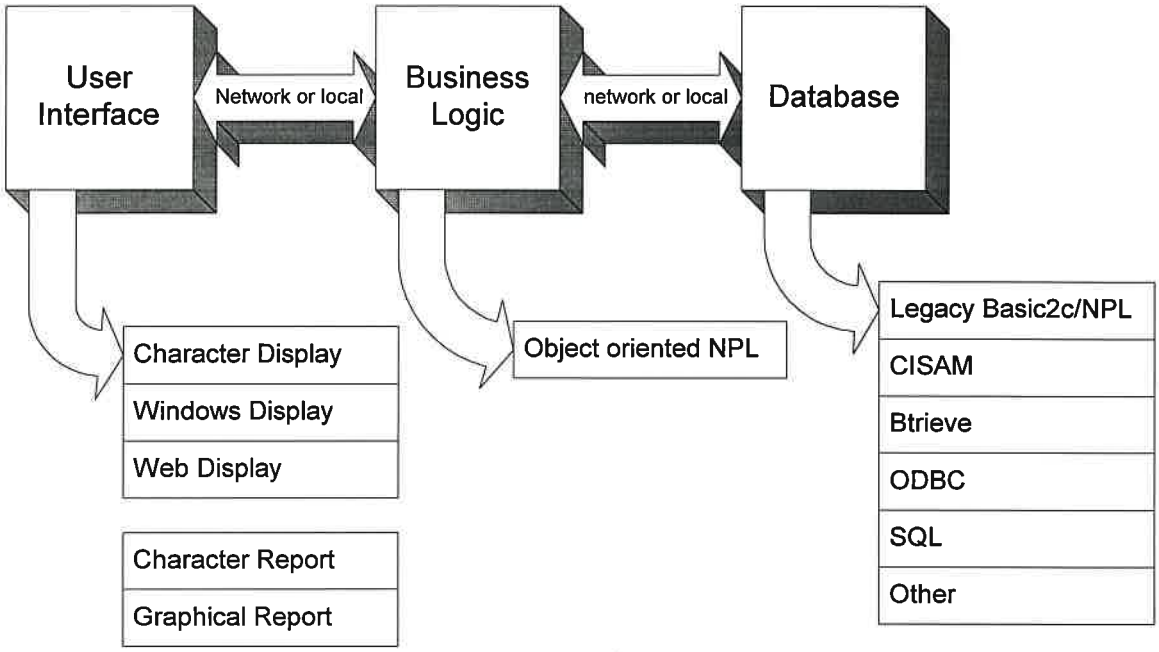
(**Sam Matzen**, document "Decision 1999") Through enhancements to the NPL Workbench and Integrated Development Environment, the application is deployed in a true multi-tiered-network environment that separates the user interface, business logic and database interface into three separate areas.

Application will operate under industry standard window managers including Microsoft Windows.

Resultant system will be structured so that it can be merged into a web based informational interface as an e-commerce application.

Will provide database interface for transaction processing.

This will be accomplished by providing the developer with the tools necessary to quickly deploy an application into this highly advanced environment:



Assuming the business application is just being re-deployed and not redesigned and deployed, the deployment process will include the following steps:

Use the workbench to:

- Model the application including the links between the user interface forms and reports. This modeling provides the basis for generating the initial windows with menu options to provide for transfer from one window to another.

- Model the database. A tool will be made available to extract much of the existing database model from the legacy application.

- Model the database constraints. A tool will be made available to extract many of the existing constraints from the legacy application. Constraints are the basis for input verification and provide the details necessary for creation of select and combo boxes.

- Model the database relationships.

- Model the business logic classes. This includes modeling the user interface and database interface properties, events and methods.

- Model the web application forms.
- Model each individual form and report. A tool will be made available to extract much of the content from the legacy application.
- Generate the application framework.
- Enhance, modify and test the application prior to test deployment.
- Model the data conversion and generate the data conversion programs. A tool will be made available to extract much of the data conversion details from the legacy application.

Deploy the application:

- In test environments and use the workbench to make modifications. Use the workbench update deployment tool to deploy updates into the text environments.
- Into the production environment.

Much of the success of this type of re-deployment depends on the amount of application modification being made during re-deployment. Because of the restrictive environment in which many of the legacy environments were developed, there may be a long list of enhancements that will be made during re-deployment.

Up Side:

Resultant application will be able to effectively deploy on all classes of user interface including windows, web and character.

Resultant application will be able to deploy in the heterogeneous computing environment including Windows 95/98, Windows 2000, Windows NT, and the Intel Unix variants.

Resultant application will be cross-platform deployable.

Resultant application will be web enabled.

Resultant application will be modern, thus reducing the reluctance of new developers to enter the NPL arena.

Completely standards compliant modern look and feel.

Niakwa will have an enterprise class application development product to sell.

Down Side:

Will require significant enhancements to workbench.

Will require some VAR training.

Can we net-enable their entire (potentially huge) application (including browser interface and transaction processing) in under a week? Probably not. There isn't enough business logic in the current applications to make this happen. Most applications don't lend themselves well to transaction and/or forms based processing. What we can do is provide the developer with tools to re-deploy his application on top of a template that will allow for all types of user interfaces, isolate his business logic, and provide access to unlimited types of databases.

Can we do this in 1999? If we don't let first release of the project creep very much we can ship in 1999. This will require some quick decision making on your part. If you wait until July, it isn't going to happen. However, I have been working on this problem for about 5 years now, and have many of the components in some form of completion and deployment. So, we wouldn't be starting from scratch.

Transform NPL

Remake NPL as a Java-Like Language

(**Tim VeArd**, phone discussion) Permit downloadable chunks of code to run on the client machine without goldkey or dongle security.

(**Craig Freeman**, e-mail message) The idea of "web enabling" NPL needs some clarification.

Web-Maintained Central Databases. With ISDN, cable MODEMs and DSL, you can get data off a server pretty fast these days. If Russ [Fairchild] modified QUASAR to build SQL commands on the fly, you could make the SPEED data manager a built-in feature of NPL and support databases which are both local and remote. The actual NPL code would execute locally but data would be stored who-knows-where on the Web.

NPL Turned Java. The entire NPL Runtime is a relatively small Windows application. You could provide a Java program for servers that would download NPL to the end user as needed and keep it and the NPL program code up-to-date when new versions are installed. In this way you could install a single "Internet" version of NPL and support any number of remote users.

This "Special" Runtime would work only after a password sign-on that would count simultaneous users on the server. And it would keep running only while connected to a licensed server. I use a similar technique in my existing NPL code for fear someone will log into a remote Novell Network, pass security via TCP/IP and then sign off to run all day locally. Every time the user exits back to the menu, I check to see if the local node is still connected to the server.

Running Actual Apps On The Server. Windows WTS and Citrix handle this nicely.

pcAnywhere For The Web. pcAnywhere supports the use of the Web for remote dialup. In this application (like with WTS/Citrix), leaving NPL apps in text mode has the advantage of greatly speeding up display updates. An outsider viewing NPL running remotely might actually think Niakwa designed NPL for just such an application.

Putting It All Together. If you combine the above with the GUI ideas [in Provide a Screen Scraper], you have a pretty powerful toolbox. For example, a florist with 15,000 remote locations could choose between Citrix running on an Alpha server or dial-up Internet TCP/IP with NPL running on the nodes. Either way, he would never encounter a broken link nor errant VB routines on the server.

Instead of facing a megabyte implementation plan, a reseller could offer to install the app like it was a single-user installation and let the user buy a gazillion Runtime license upgrades (I would love to see the diskette labeled "Licensed for 16,384 Users on Approved Networks".)

Make NPL an Object-Oriented Environment

(**Donovan Burkhart**, phone discussion) NPL needs to provide an object-oriented development environment with all necessary core functionality. Otherwise developers will have to rewrite (and why in NPL?)

(**Sam Matzen**, phone discussion) Long term: Don't just band-aid a dinosaur, but rather solve the underlying problem:

> Isolate legacy logic into objects.
>
> Extract data management into NDM.
>
> Throw out user interface code and rewrite.

(**Mike Liston, Sam Matzen, Pat Legg, Alan Green**, Object discussion 12/97) At the instigation of Mike Liston, we held a conference call to discuss the topic of objects in NPL. It was a very productive discussion and we came to these conclusions:

> Object technology is now a part of all leading languages (VB, C++, Java, etc.)
>
> NPL makes peripheral use of objects through its integration with VB in Visual NPL.
>
> NPL has the capacity to become an object-oriented language in its own right. The questions are "Should we?" and "How should we?"
>
> In terms of how NPL should evolve into an object-oriented language (if that's what we all decide should happen), a two-step process seems desirable:
>
> > 1. In the short term, extensions should be added to the language to begin to take us towards object orientation.
> >
> > 2. In the longer term, we should aim NPL towards a larger goal of object compatiblitiy both internal to the language (NPL objects) and outside the language (data exchange with and manipulation of objects in external environments such as C++. And we should build on the extensions we

developed in Step 1 (which means that we must be forward-thinking when we decide on their nature.)

Two possibilities suggested themselves for adding object orientation to NPL (and there may be more):

1. Expand NPL modules as objects. After all, they already incorporate some object qualities: local data which can be hidden or exposed to outside routines, and local routines (object methods) which can be triggered from the outside.

2. Use and expand on the existing GenFive technology. GenFive is a system from SLM software (Sam Matzen) which relies heavily on the ability of NPL to generate and then load modules at runtime (read: create and then execute methods of objects).

I should digress here a moment to explain what objects are:

Objects are self-contained constructs holding both data and program code.

Once an object of a given type has been built (a class of objects has been created), new objects of that type can be created. These copies are called instances of the class.

Objects have properties, which are the attributes of the object. For example, a visual object such as a VB text box has the property "color".

Objects also have methods, which are blocks of program code contained within the object and callable by name from the outside by entities which have been given access to them.

Objects also exist in an event-driven paradigm. Events are occurrences that, when they happen, trigger the execution of some program code within the object. Example: an accounts-receivable object might execute some code to update a customer file when a payment event is triggered. Events happen in real time, and differ from the procedural direction of most program flow by allowing code to branch to other objects and events on an ad hoc basis.

And there's more. For example, new object classes can inherit properties from their predecessors.

But you get the idea. Objects are a new game for many NPL'ers. But surprisingly, many of you have been taking advantage of some object techniques for a long time. One example immediately comes to mind: SPEED and its successor, FourD, have always been

event-driven (as opposed to procedural) in the way users could navigate through the system.

(**Pat Legg**, e-mail message 12/97) NPL and objects.

The long term objective for objects and NPL would be to introduce a formal CLASS syntax, and syntax for declarations of appropriate objects, and access to the members and properties. Due consideration will be given to inheritance issues and performance goals.

Sure, fine, but what about right now? An NPL programmer that wants to have object-like programming capabilities basically has(almost) what is needed to get the job done, provided you don't mind some extra keyboard work and some pretty ugly syntax. How would you do this? Well you could read Sam Matzen's white paper but the bare details are:

> A class is a PUBLIC RECORD.
>
> An object of the class is a string declared to be the length of the record.
>
> Members of the class are FIELDs of the record. The field name is always prefixed with the class name to ensure uniqueness.
>
> Methods of the class are public functions. The name of the function is always prefixed with the class name to ensure uniqueness.
>
> To really ensure uniqueness, a class name should have no embedded '_'s and probably starts with the initials of the implementor.
>
> Every method must always have as the first parameter a /POINTER to a string 'this$' which is assumed to be a record of the appropriate class.
>
> The object constructor and destructor functions must be called explicitly before using the object and after it is no longer in use.
>
> The record definition and methods dealing with the record are collected in a single NPL module. Any application that wants to use objects of the class must INCLUDE that module.

(**Mike Liston**, e-mail message12/97) I think NPL classes would be the greatest leap forward for the language since MODULEs. I have used VB classes in my Visual NPL development and found them to be quite handy. And while I'm not sure how I might apply classes to my NPL work, I didn't know how to apply MODULEs three years ago, either.

It also seems to me that explicit support for classes would also make NPL much more attractive to new developers.

(**Sam Matzen**, e-mail message 12/97) Objects in NPL. I agree completely with Pat's thoughts on NPL objects.

Object technology is a way of looking at and describing software. The terminology provides a standard term set for describing the relationship between various object components. You can program NPL 4.22 using objects and describe the software using object terminology.

NPL as a whole is quite compatible with object technology, however, there could be some syntax enhancements (as Pat has pointed out) that could simplify object implementation.

The area of greatest concern is with memory management, or the creation and destruction of instances of objects. I have implemented objects using MAT REDIM.

After working with object technology in NPL for almost four years, I feel that objects are the only way to go. This technology simplifies the development of complex interactive multi-instance applications. The implementation of both local-memory classes and persistent data classes encapsulates, simplifies and standardizes memory management.

Middleware Layer

Provide Middleware that GUI-izes and Net-Enables

(**Chris Maxwell**, phone discussion) Another possible solution, for developers with 100+ installs: help them port their apps to FourD, where they get standardized logic, GUI and open data.

(**Russ Fairchild**, phone discussion) Instead of trying to develop an automated tool, we should provide a methodology and education in its use.

FourD has that methodology, as well as good utilities and up-to-date technology. We can solve this problem by converting legacy apps to FourD.

(**Chris Cummings**, e-mail message) My initial conclusion is that there are TWO hurdles with existing NPL apps: their non-GUI look, a formidable problem, but more important, their non-GUI behavior, which is integral to the app's design. We can come up with all sorts of ways to GUI-ize their appearance, but as you've apparently concluded with Vinnie, and as we experienced with SideFX, procedural apps with character-based screen organization become, at best, mediocre Windows apps, and Don and I had pretty good material to start with. Trying to generalize the conversion of a typical NPL design to an event design with the time frame and effort you mention for most of the conceivable things (many of them bad!) people do with NPL is just beyond my comprehension. I won't stop thinking about it, but I'm not sure right now where to begin chipping away at it.

The second conclusion is that if GUI/browser is where you want to go (and I think it's a good direction), Niakwa's going to have to play a bigger role than maybe you currently imagine. There are too many problems, and if you leave more than a couple to the VAR to solve, it just won't happen, at least with NPL.

When I first got into programming, I developed this little F&I app for Wang calculators. Working down in machine language taught me a lot about organization and efficiency such that by the time I ported to Wang BASIC, I could do more in less space and in less time than anybody I knew. I had a good app, and my customers were happy. I was probably in a position similar to many of your VARs. I thought about adding new capabilities, but I was constantly stymied by a lack of stuff I could learn from. Then in 1978, I ran into this middleware product called SPEED. Zounds!

SPEED was not an easy transition. Many of the concepts were just beyond anything I'd experienced. I'd code for a few minutes, then go read the manual to try to figure out what to do next. The docs told me what SPEED did and how to use the components, but didn't tell me how to design a SPEED app. I persevered, but the turning point was when I got a

copy of TOM's D1 distributor apps, and the blinders came off. This was a REAL app, not sample code. I could see how SPEED was meant to be used. It was the best education I ever had. And with SPEED providing this neat toolbox, this ol' DATA-SAVE-DC coder was able to quantum jump to a new level of application design. Without it, well, it would have taken me a while longer....

So the rough parallel is this: Many of your VARs are steeped in "DC" BASIC. You hand one Vinnie as if GUI-izing an app is something he knows how to do, when actually he doesn't even know where to start. Vinnie seems to be meant for NPLers who don't know VB. But if they don't know VB or something like it, how are they going to learn to think event-like so they can use Vinnie?

The answer that you wanted from me was some way to take existing NPL apps, and through some imaginative technique, transform them in a few man hours to GUI/DHTML-enabled apps. If it's possible, I think I would understand the solution when I saw it, but right now I not only can't imagine it, I don't think the results would be marketable. I don't see how programmer/authors are going to avoid having to rewrite their apps, and I know that's a concern – if they have to rewrite, why do it in NPL? So, my thinking goes after THIS issue. What would it take for an NPL author to want to rewrite his app for graphics AND stay in NPL?

Borrowing a couple pages from TOM, it's going to take some middleware, some generic apps, and some changes to the language. You need a "SPEED" for GUI/Net that the VARs can transfer their vertical expertise to, and some horizontal apps that they can verticalize, such as direct net sales, secure data exchange, image management, an e-mail interface, etc., and yes, general accounting would also be nice to get people started. You're still in the language business, but it's going to take more than just the language to get people moved, trained, and successful in this new world.

If I'm a VAR, and I can easily put together a GUI app in NPL that's both useful and looks good, that minimizes the effort required to convert my existing customers, that gives me access to e-commerce, and that is open ended enough and supported enough to keep me from falling behind again, you have my loyalty for the next decade. You also have some additional products to sell.

It will also be expensive.

So the gamble is this. NPL is the best prototyping language on the planet. When it comes to creating something logically new and non-trivial out of wholecloth, a programmer can do most things faster and easier in NPL than in any other language (and when it's done, performance isn't too shabby either). Instead of just nursing the VARs along, can we use NPL not only to gain access to the GUI/Net world, but also to compete successfully in it? (Details at 11 on NewsWatch.)

Technically, I think the answer is "Yes", but unlikely in 1999.

(**Chris Cummings**, phone follow-up) We should provide some middleware, some generic apps, and some changes to the language. In other words, we need a GUI-ized, Net-enabled SPEED-like system that VARs can transfer their vertical expertise to, and some horizontal apps that they can verticalize.

This approach also provides Niakwa with some additional products to sell.