Contents

Software Manufacturing:

The Industrial Revolution of Software Development

A cover story in the September 1994 issue of Scientific American is entitled *The Crisis in Software*. As might be expected, the lead horror story involves the software debacle in the baggage-handling system at the new Denver airport. Antediluvian software engineering practices are to blame, we are to believe. “Unfortunately, the industry does not uniformly apply that which is well-known best practice,” quotes Scientific American of Larry E. Druffel, director of Carnegie Mellon University’s Software Engineering Institute. In other words, if we could cure our engineering ignorance we would eliminate our software misadventures.

In the article, Brad J. Cox suggests that “It’s like musket making was before Eli Whitney. Before the industrial revolution, there was a non-specialized approach to manufacturing goods that involved very little interchangeability and a maximum of craftsmanship. If we are ever going to lick this software crisis, we’re going to have to stop this hand-to-mouth, every-programmer-builds-everything-from-the-ground-up, preindustrial approach.”

Software Manufacturing

Exactly. But it will take more than better engineering practices. Much more.

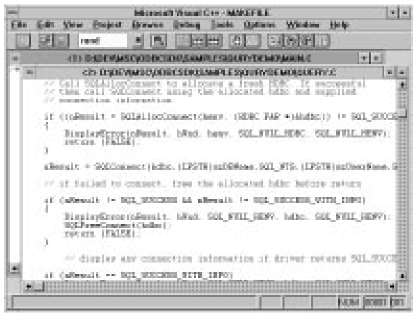
Astonishingly, Scientific American fails to recognize the role that machine tools played in the industrial revolution. Rifle parts became interchangeable because machines made them identical.

The industrial revolution was the outgrowth of a great number of visionaries who believed that better tools would produce better products cheaper and faster. If we are to make better software, cheaper and faster, we will need better tools. To do so, we need to understand the weaknesses of the tools we are now using.

This white paper was written by Bruce D. Barrington, CEO of TopSpeed Corporation and author of the Clarion language and Clarion Template language. Richard Chapman of the TopSpeed Development Centre also contributed to this paper.

Tools for Software Craftsmen

Is there any doubt left in anyone's mind that Microsoft Windows has become the desktop standard for stand-alone as well as client software? What tools are available to design, express, and deploy graphic user interfaces?


Microsoft Visual C++ - MAKEFILE
File Edit View Project Browse Debug Tools Options Window Help

```
// Call SQLAllocConnect to allocate a fresh HDBC. If successful
// then call SQLConnect using the allocated hdbc and supplied
// connection information.

if ((nResult = SQLAllocConnect(henv, (HDBC FAR *)&hdbc)) != SQL_SUCCE
{
    DisplayError(nResult, hWnd, henv, SQL_NULL_HDBC, SQL_NULL_HSTMT);
    return (FALSE);
}

nResult = SQLConnect(hdbc,(LPSTR)szDSName,SQL_NTS,(LPSTR)szUserName,S

// if failed to connect, free the allocated hdbc before return

if (nResult != SQL_SUCCESS && nResult != SQL_SUCCESS_WITH_INFO)
{
    DisplayError(nResult, hWnd, SQL_NULL_HENV, hdbc, SQL_NULL_HSTMT);
    SQLFreeConnect(hdbc);
    return (FALSE);
}

    // display any connection information if driver returns SQL_SUCC

if (nResult == SQL_SUCCESS_WITH_INFO)
```

An OOP language at work: Microsoft Visual C++

They fall into two categories: visual tools and object-oriented languages. Visual tools such as Visual Basic and PowerBuilder maintain Window layouts in a private repository and process them with an internal messaging engine. An application's behavior is produced by associating "snippets" of event processing code with the controls in a Window. This strategy produces an inherent scaling problem as large applications become "hidden behind a thousand doors". The fact that "code snippets" are interpreted, rather than compiled into machine language, makes all visual applications sluggish. Large visual applications can be unbearably slow. Furthermore, developers often "hit the wall" erected around an internal messaging engine that often fails to export the fine control necessary to implement a complex specification.

Object-oriented languages such as C++ provide the necessary performance and control at the cost of a daunting and esoteric syntax. It is a fact that C++ doesn't "know" anything about Windows. It must be "taught" the entire Windows class library on every compile. Then C++ promptly forgets what it learned before the next compile. Language elements that COBOL programmers have taken for granted for 30 years must be "inherited" by every C++ source module. It is no wonder that C++ is complicated. Instructing a compiler about its basic grammar is tricky business indeed. As a result, conventional programmers cannot even read a C++ program without a lot of training.

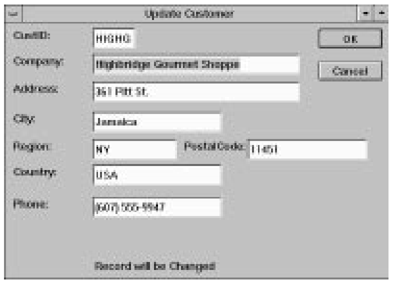
Objects of our Affection

We need GUI languages. Fifteen years ago, the relational database revolution launched the entire database tool industry into a frenzied effort developing fourth generation languages. The idea was that an elegant and versatile database grammar makes database programming easier and database programs more reliable. Today, few would argue with that premise. So where are the GUI languages? Where is the elegant and versatile user interface grammar that makes coding user interfaces as easy as accessing databases with a 4GL. Has the entire world bought into the notion that object-orientation is the last word in GUI programming?

Let's hope not. Because if we have learned anything in the object-oriented age, it is that objects don't scale! The truth is that OOP has reneged on its promise to deliver reusable custom components that can be assembled to create applications without programming.

Intermediate objects, like Windows controls, are very successful reusable components that are easily deployed with little or no programming. But big custom components like forms, reports, updates, calculations, etc. are still hand-coded. And these components must be stitched together with more hand-code to create an application. Unfortunately, the process of writing code—even object-oriented code—hasn't changed much since the introduction of COBOL.

Objects are hand-crafted parts. Worse than that, an object is an extremely primitive model for an entire application or a large custom component of an application.



CustID:	HIGHG
Company:	Highbridge Gourmet Shoppe
Address:	361 Pitt St.
City:	Jamaica
Region:	NY
PostalCode:	11451
Country:	USA
Phone:	(607) 555-5947

Record will be Changed

A typical database update form.

Consider a form that updates a database: An update form needs hundreds of properties to describe its window layout, database access strategy, referential integrity constraints, etc. How do you manage them? With a spreadsheet that is two columns wide and a thousand columns long? Remember that properties interrelate. If you set one property you can't set another. Or perhaps you must set it. Where does an object keep its property rules? When and where does the object report a conflict?

Managing properties isn't the only problem with big objects. Properties represent features that must be inherited. That means the feature set of a big object must be arranged in a hierarchy or be inherited in its entirety. Unfortunately, feature sets resist any effort to sequence them. Which comes first: VCR controls or auto-increment keys?

The answer is "neither." Those features are not related in any meaningful way. Feature sets aren't hierarchies. That's why big objects come in one version—complete with all the features a designer can dream up. Big objects aren't just a little bigger than their parents. They are enormously bigger. Swallowing these objects whole when only a nibble would suffice is a primary contributor to the object bloat that is now being called the "fat client."

We need reusable design objects. With the infrastructure necessary to gather and maintain a complex set of inter-related properties. So each design object can have its own custom user interface to make sense out of its repertoire of features.

We need design objects to automatically generate the code necessary to "glue" applications together and to produce the behavior expected of large components. Design objects that propagate only the functionality requested or that is required to resolve issues between interrelated properties. So applications can be as small as they ought to be.

We know what we need but what have we got? For now, we are building client software with hand tools. Like a cabinet maker armed with a cold chisel and a scalpel, we must decide between Visual Basic or Visual C++. PowerBuilder or SmallTalk. We are stuck with an object model that won't grow any bigger than a data bound control. We hand craft all behavior that doesn't come out of a VBX.

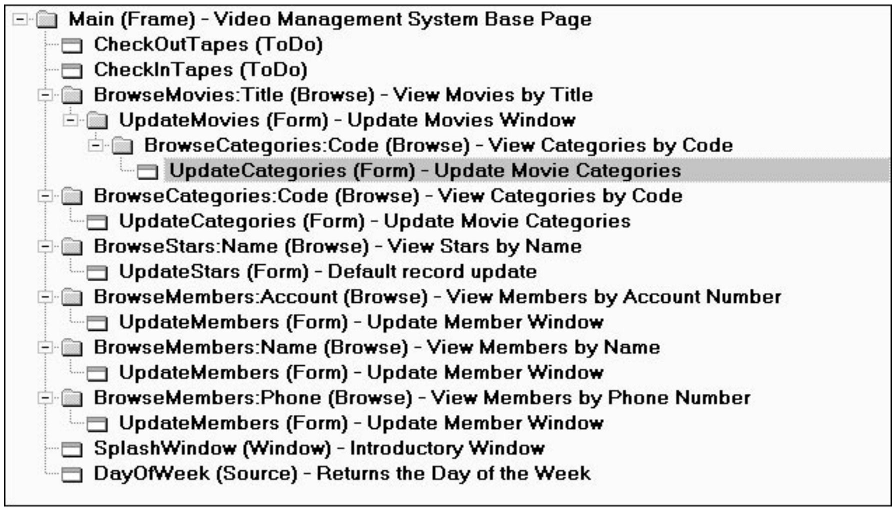
There is no abstraction to represent an entire application. No application model. No application repository. There is no large component model. No standard large components other than the standard Windows dialog boxes. We design applications then we write them and test the code. Every new project represents the same business risk as the last. We are artisans trapped in the renaissance. The quality of our work depends on our talent.

The Post-industrial Software Age

Or so it would seem to many. Fortunately, some developers know differently. For the last 10 years, TopSpeed Corporation has been chipping away at the software crises with a steady stream of new software development technology in a search for a cure to the three primary diseases of pre-industrial software: poor productivity, poor performance, and poor reliability. Their quest has been successful.

TopSpeed has created an "application model," that treats an application as a single entity rather that a disconnected set of forms, reports, and subroutines. The application model is a container for procedure definitions that are related logically and physically. Logically, procedures are represented by a tree structure that shows calling sequences. The top procedure of the application tree is the first procedure called by the application. Below are the procedures called by the first procedure followed by the procedures they call, etc. Physically, procedures are grouped into source modules that will receive generated code.

The TopSpeed Application Model



```
Main (Frame) - Video Management System Base Page
  CheckOutTapes (ToDo)
  CheckInTapes (ToDo)
  BrowseMovies:Title (Browse) - View Movies by Title
    UpdateMovies (Form) - Update Movies Window
      BrowseCategories:Code (Browse) - View Categories by Code
        UpdateCategories (Form) - Update Movie Categories
  BrowseCategories:Code (Browse) - View Categories by Code
    UpdateCategories (Form) - Update Movie Categories
  BrowseStars:Name (Browse) - View Stars by Name
    UpdateStars (Form) - Default record update
  BrowseMembers:Account (Browse) - View Members by Account Number
    UpdateMembers (Form) - Update Member Window
  BrowseMembers:Name (Browse) - View Members by Name
    UpdateMembers (Form) - Update Member Window
  BrowseMembers:Phone (Browse) - View Members by Phone Number
    UpdateMembers (Form) - Update Member Window
  SplashWindow (Window) - Introductory Window
  DayOfWeek (Source) - Returns the Day of the Week
```

A procedure definition contains the specification for the visual elements such as menu, window, and report layouts. In addition, the procedure definition specifies the expected behavior of the procedure.

The application model and procedure definitions are maintained in an application repository. When combined with a data dictionary, the application repository can be manufactured into a complete, functioning application with a single mouse click.

The enabling technology is a powerful template language that controls a high speed source code generating engine. The template language also manages the development environment and provides a user interface for maintaining properties. TopSpeed calls these reusable design objects "rich templates."

All data elements in the application repository and data dictionary are exported as symbols available for template processing. Rich templates use these symbols to generate procedures, controls, and embedded source code. Rich templates also control the process of creating source modules and produce documentation files. Rich templates can even be used to modify other existing templates. All logic is embodied in the template language making the application generator tool infinitely extensible.

Rich templates are TopSpeed's cure for big object disease. Like objects, rich templates are reusable. Unlike objects, however, a rich template has a "front end" to gather and maintain properties.

A rich template that produces a "browse" procedure may require a half-dozen custom dialog boxes to specify its functionality. Most of these properties define optional behavior. For example, do you want to edit the records in place or use an update procedure? If so what is the name of the update procedure? Do you prefer a vertical scroll bar or VCR buttons?

These properties are organized into a convenient user interface explaining the behavior they produce and the rules governing their usage. Only the code necessary to implement the requested behavior is generated by a rich template. Each feature is customized to be compatible with other selected features.

The TopSpeed Technology Stream

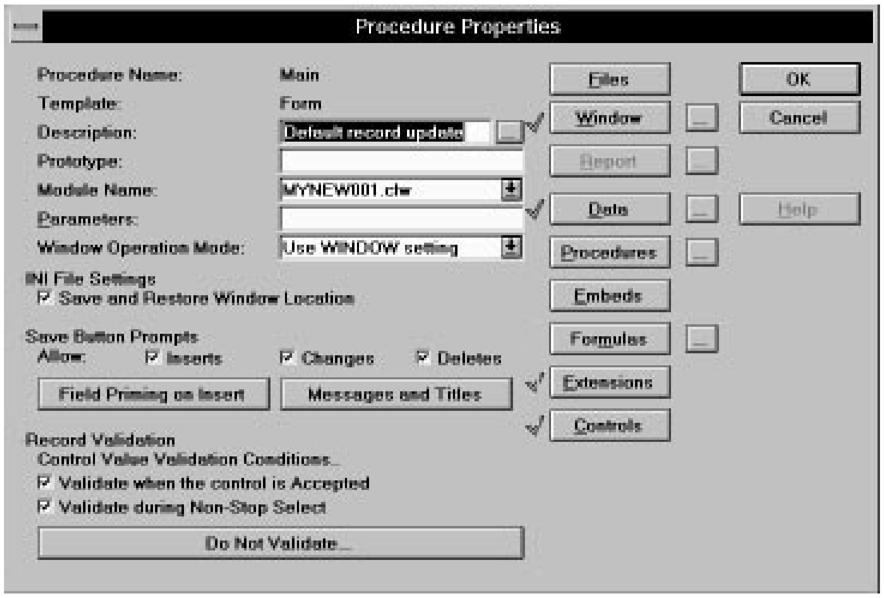
1986	**The Clarion language - a 4GL with data abstractions for screens, reports, and files along with an integrated messaging model for the user interface.**
	"Two-way" screen and report painters.
1988	**Software manufacturing - a complete application automatically created from an application repository and a data dictionary.**
1990	**Multi-language development platform - Pascal, Modula-2, C, and C++ sharing the same optimizing compiler.**
	Object oriented extensions for Pascal and Modula-2.
	"Smart-linking" - all unreferenced code and data eliminated from EXEs.
1992	**Extensible Software Manufacturing - rich templates control the process of creating applications.**
	The Clarion optimizing compiler - a 4GL matches the performance of C compilers.
	Scalable database drivers - SQL and navigational databases accessed with native APIs.
	"Smart method" linking - unreferenced virtual methods eliminated from EXEs.
1994	**"Toolware" for Software Manufacturing - rich templates for procedures, controls, embedded source code, module generation, and other design processes.**
	Windows hosted debugger with automatic redraw services.
	Window declarations and messaging model for the Clarion language.

Extensible "Toolware"

The following page illustrates the properties dialog box for a "Form" template that generates record update procedures. The right column of buttons launch interactive tools that create database schemas, paint Window and Report layouts, collect embedded source code, etc. The remaining fields gather the other information required to generate source code for a custom form. Buttons on the left of the window, such as "Messages and Titles," display dialog boxes that gather additional information. The format for each of these dialog boxes is contained in the "Form" template.

Importantly, rich templates don't replace objects. On the contrary, rich templates can be very useful organizers for objects. For example, rich templates have been "wrapped" around VBX custom controls to provide an attractive user interface, generate property assignment statements, and bind the VBX into the Clarion messaging model.

The Form Template User Interface



Procedure Properties

Procedure Name: Main
Template: Form
Description: Default record update
Prototype:
Module Name: MYNEW001.clw
Parameters:
Window Operation Mode: Use WINDOW setting

INI File Settings
☑ Save and Restore Window Location

Save Button Prompts
Allow: ☑ Inserts ☑ Changes ☑ Deletes

Record Validation
Control Value Validation Conditions...
☑ Validate when the control is Accepted
☑ Validate during Non-Stop Select

Rich templates are not even software. They do not contain instructions for a computer to follow. Rather, rich templates instruct the software manufacturing tool how to communicate with the designer and how to fabricate his design. TopSpeed calls this "toolware." New toolware is continually produced by TopSpeed and third-party developers alike, delivering a constant stream of new functionality to TopSpeed developers.

A "Gooey" 4GL

Windows programming should be simple. The user interface is so completely standardized that very little detail is necessary to specify the "look and feel" of a program. However, the natural purity inherent in the Windows graphic user interface has been contaminated by a set of inelegant tools that have been adopted as industry standards.

The following page lists the contents of four files required by Microsoft Visual C++ to specify a "Hello World" program. All the comments and compiler directives that control the Visual C++ environment have been removed to reveal the "essence" of a minimal C++ application that uses Microsoft Foundation Classes. Microsoft created its Foundation Class Library to simplify Windows programming under C++. It is hard to conclude that they have succeeded.

Reading even a simple Visual C++ program like this requires a working knowledge of the syntax of both C++ and Windows resource scripts along with a basic understanding of the formidable Foundation Class Library. To the casual observer, or for that matter, to a well-trained conventional programmer, "Hello World" may as well be written in hieroglyphics.

```
//Contents of HELLO.H

#ifndef _——HELLO—H——__
#define _——HELLO—H——_

class CMainWindow : public CDialog
   {
   public:
        CMainWindow();
        afx_msg void OnOkButton();
        DECLARE_MESSAGE_MAP()
   };
class CTheApp : public CWinApp
   {
   public:
        BOOL InitInstance();
   };

#endif _

//Contents of HELLO.CPP

#include “stdafx.h”
#include “resource.h”
#include “hello.h”

CMainWindow::CMainWindow()
   {
   Create( “HelloBox”, NULL );
   }
   void CMainWindow::OnOkButton()
   {
   CloseWindow();
   }

BEGIN_MESSAGE_MAP( CMainWindow, CDialog )
   ON_COMMAND( IDM_OKBUTTON, OnOkButton )
END_MESSAGE_MAP()

BOOL CTheApp::InitInstance()
   {
   TRACE( “HELLO WORLD\n” );
   SetDialogBkColor();
   m_pMainWnd = new CMainWindow();
   m_pMainWnd->ShowWindow( m_nCmdShow );
   m_pMainWnd->UpdateWindow();
   return TRUE;
   }

//Contents of RESOURCE.H

#define IDM_OKBUTTON                     100

//Contents of HELLO.RC

#include “resource.h”
#include “afxres.h”

HELLOBOX DIALOG DISCARDABLE 34,22,144,75
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
      WS_SYSMENU
CAPTION “Visual C++”
FONT 8, “Helv”
BEGIN
   CTEXT “Hello World”,IDC_STATIC,0,23,144,8
   DEFPUSHBUTTON “OK”,IDM_OKBUTTON,
56,53,32,14,
WS_GROUP
END
```

A Visual C++ "Hello World"

Contrast the Visual C++ “Hello World” with its Clarion equivalent immediately below the C++ example. This program can probably be understood by anyone familiar with Windows, whether or not they have ever written a computer program.

PROGRAM introduces the statements that declare data elements. WINDOW declares a “Clarion for Windows” dialog box named *Window*. *Window* has a system menu, measures 160 dialog units wide by 60 units high, and is centered (because the x and y coordinates are omitted). *Window* also contains a text string and a default button named *?OK*.

CODE introduces statements that open and process *Window*. ACCEPT... END is an integrated messaging loop that cycles for all window events. When the *?OK* button is pressed or when the system menu closes the window, the program breaks out of the ACCEPT loop and returns to Windows.

Is it fair to compare the Clarion language with C++? Can Clarion really match the performance and power of C++?

Clearly, Clarion matches the performances of many C/C++ compilers. Clarion shares the same optimizing code generator with TopSpeed Pascal, Modula-2, C, and C++ compilers. As a result, Clarion object modules are indistinguishable from C++ object modules. As measured by the standard benchmark known as the “Sieve of Eratosthenes”, Clarion is 20% slower than Microsoft Visual C++ when both are optimized for speed. Currently, Visual C++ holds a similar performance edge over other C/C++ language products.

The same benchmark shows Clarion to be 36 times faster than Visual Basic and a breathtaking 1,300 times faster than PowerBuilder. This difference can be explained by the fact that both Visual Basic and PowerBuilder produce scripts that must be *interpreted* at run-time. C++ and Clarion produce native code executed directly by the computer.

Development System	Cycles
Microsoft Visual C++ 1.5	136,397
Clarion for Windows	**106,667**
Microsoft Visual Basic 3.0	2,594
Powersoft PowerBuilder 3.0a	82

Of course, Clarion is not as powerful as C++ for low-level systems programming. A Clarion program can directly call C/C++, Pascal, and Modula-2 if that is necessary. Because of TopSpeed’s multi-language support, these languages can be freely mixed in the same project.

```
   PROGRAM

Window  WINDOW(‘Clarion for Windows’), AT(,,160,60), SYSTEM
        STRING(‘Hello World’), AT(30,15,90,12), CENTER
        BUTTON(‘OK’), AT(60,35,,), USE(?OK), DEFAULT
        END

CODE

   OPEN(Window)
   ACCEPT
      IF ACCEPTED() = ?OK THEN BREAK.
   END
   RETURN
```

A Clarion "Hello World"

Clarion for Windows itself is a mixed language project. The Clarion run-time library, the Clarion compiler, the Clarion source generating engine, and the data dictionary editor are written in C++. The debugger and the optimizing code generator are written in Modula-2. But the entire user interface is written in Clarion. The database manager that accompanies Clarion for Windows is also written in Clarion. Using Clarion everywhere possible substantially reduced the development effort required to produce Clarion for Windows.

The Clarion language is a general purpose business language with built-in data abstractions that simplify user interfaces and database access. For developing client or stand-alone vertical applications, Clarion is clearly superior to C++.

To summarize the benefits of the technology introduced by Clarion for Windows:

- **Clarion's rich templates eliminate most of the hand-coding required by other visual tools.**
- **Clarion's performance matches C++ and far exceeds the other visual tools.**
- **The Clarion language simplifies the process of developing, testing, and maintaining Windows applications.**

The conclusion is inescapable: If you can use Clarion, you *should* use Clarion.

Software manufacturing techniques now produce more reliable applications in less time using fewer resources. The post-industrial age of software development has arrived.

The TopSpeed Legend

Niels Jensen

Bruce D. Barrington

In 1992, Clarion Software Corporation merged with Jensen and Partners, International to form TopSpeed Corporation. The principals involved were Niels Jensen, CEO of JPI and Bruce Barrington, CEO of Clarion.

Jensen is the founder of Borland, International and is the visionary behind Turbo Pascal, the first integrated development environment. In 1988, Jensen and the entire language development team left Borland as a group in a dispute over compiler quality. JPI purchased their work in progress and produced the TopSpeed line of compilers which immediately became the performance leader of the desktop languages market. Later, the merged companies would adopt the TopSpeed trademark as their corporate objective as well as their company name.

Barrington is co-founder of HBO & Company, now a $300 million information services company serving the health care industry. In 1983, Barrington founded Clarion Software Corporation. Barrington is the author of the Clarion language and the Clarion Template Language.

Barrington is now the CEO and largest stockholder of TopSpeed Corporation. Niels Jensen is the second largest stockholder and a member of the TopSpeed board of Directors. The JPI development team, headquartered in London, has been expanded and is now called TopSpeed Development Centre.