Bridging traditional and object technologies: Creating transitional applications

by A. H. Lindsey
P. R. Hoffman

Object technology is a well-known advance for developing software that is receiving a great deal of attention. Unfortunately, the educational investment required and the additional complexity introduced by most tools that support this technology have dampened its rate of adoption by many enterprise developers. To bridge this skills and technology gap, development tool strategies are presented that encourage a more evolutionary approach, easing the transition. Rather than requiring totally new skills and tools, these strategies take advantage of the strengths and familiarity of traditional facilities—they hide much of the raw technological complexities and yet exploit the strengths of object technology by supporting the creation of transitional applications. The strategies described fall into two categories: bridging between the traditional and object worlds, and masking the complexities of object technology by exploiting higher-level rapid application development techniques.

Object technology,[1] a software technique that has been well known within the research and advanced development communities for many years, has recently begun to receive attention in commercial environments. This attention has resulted in inflated productivity and cost-savings claims, numerous product renamings (to add the "OO" badge for "object-oriented"), and a never-ending stream of articles, books, and seminars on all facets of this technology. While nothing could deliver all the benefits that the hyperbole suggests, object technology appears to have enough momentum to be a major force in the software arena for many years.

Unfortunately, the exploitation of object technology has been difficult for many enterprise development organizations. This difficulty can be attributed to many factors, but the most prominent is a widespread lack of skills. Most object technology practitioners believe that there is a real and nontrivial paradigm shift that must occur when moving from the procedural world into the object world. In his widely read book on object technology,[2] Taylor defines a *paradigm* as:

> An acquired way of thinking about something that shapes thought and action in ways that are both conscious and unconscious. Paradigms are essential . . . , but they can present major obstacles to adopting newer, better approaches.

He then defines *paradigm shift* as:

> A transition from one paradigm to another. Paradigm shifts typically meet with considerable resistance followed by gradual acceptance as the superiority of the new paradigm becomes apparent.

0018-8670/97/$5.00 © 1997 IBM

Object-oriented technology is regarded by many of its advocates as a paradigm shift in software development.

Because most enterprise developers are not currently skilled in object technology, this required paradigm shift is one of the major inhibitors to its adoption.

Many new tools have emerged in the last few years designed to exploit object technology. These tools, such as VisualAge* for Smalltalk[3] and VisualAge for C++*, generally require that an object-oriented language be learned. Thus, due to the paradigm shift, most of these tools are not easily used by most enterprise developers.

This paper proposes an approach that eliminates many of the obstacles just described. Rather than requiring enterprise developers to learn new analysis and design techniques and a new programming language, we assert that most of the benefits of object technology can be realized more quickly by exploiting a combination of new and traditional technologies.

Visual programming, as implemented in the "construction from parts" framework within the VisualAge family of products,[4] appears to be usable by developers with varying backgrounds, including enterprise developers. To extend their applicability, traditional languages can be minimally altered to make their capabilities fully accessible within this visual object technology context and to support part and object scripting.[5] Languages that use a procedural paradigm, including fourth-generation languages (4GLs), do not require the paradigm shift required by object-oriented languages. As Goetz states, "COBOL and other programmers can learn to use these new 4GLs in a matter of days, unlike object-oriented programming, which requires months of training."[6] Furthermore, significantly higher productivity can be achieved by hiding much of the initial and ongoing complexity associated with object technology. By exploiting rapid application development (RAD) techniques, such as data-driven template technology, much of the manual visual and procedural programming and the complex system infrastructure programming can be eliminated or hidden. In combination, these techniques remove many of the inhibitors to object technology that enterprise developers experience today. Therefore, the resulting hybrid or *transitional*[7] applications can exploit the strengths of visual programming for the graphical user interface (GUI) while maintaining the scalability, maturity, and familiarity of traditional languages for business logic, data access, and transaction processing.

The remainder of this paper describes several tool strategies that we believe make it easier to use object technology. We propose:

- Introducing *traditional parts* within the construction from parts architecture
- Adding *part and object scripting* capabilities to traditional languages
- Exploiting *RAD templates* to automatically create complete transitional applications (GUI, client, and server logic parts, as well as the associated visual links) tailored to customer data specifications

We assert that most of the benefits of object technology can be realized more quickly.

Finally, because VisualAge Generator*[8] and BW/Wizard**[9] support most of these techniques, we will use them throughout for illustrative purposes.

Bridging between the traditional and object technology worlds

Object technology, whether or not explicitly identified as such, has been around for about 30 years. Most technologists trace its roots to Simula 67 and early versions of Smalltalk.[10–13] Regardless of this long history, object technology has only recently begun to gain widespread awareness and acceptance. No longer is it a technology reserved for academics and researchers and running on only a few specialized machines. Rather, there now exist many robust development environments, supporting languages such as Smalltalk, C++, Java**, and Eiffel, that are available on most of the popular platforms. These development environments typically include browsers for viewing and editing source code, debuggers for detecting and correcting problems, and compilers and linkers for turning the source into executable code. Some of the environments even include or can exploit "object-aware" code repositories, con-

figuration management tools, version control tools, and distribution services.

One of the technologies that has helped accelerate the acceptance of object technology in recent years is the construction from parts form of visual programming.[14] Virtually all construction from parts environments support WYSIWYG (what you see is what you get) layout of the GUI controls, a technique that has existed for several years. However, these environments go further by eliminating the need to write some of the code. Instead, the developer connects parts together with visual links to trigger user interface actions, run logic, and move data. The fact that many new integrated development environments (IDEs) contain a construction from parts facility indicates that this technique is gaining interest as an alternative to textual coding for GUIs and their associated logic. Some tool vendors and industry articles even suggest that nonprogrammers can create complete applications simply by linking parts together (the GUI and non-GUI elements).[15]

While the goal of developing programs without writing any code is desirable and worth pursuing, reality and the current state of the reusable parts market indicate that trained programmers need not worry about losing their jobs any time soon. However, visual programming is a technology that is as applicable to the programming community as it is to the world of nonprogrammers. Especially given the recent push toward client/server and World Wide Web architectures, more and more new applications will require a graphical interface. Programmers can expedite the creation of applications by using one of the many construction from parts environments for the GUI,[16] allowing more time for the application architecture, client and server business logic, database design, etc.

Unfortunately, the object-oriented tools that support this form of visual programming are disjoint from the mainstream programming world. When creating the client and, if supported, server logic with these tools, the programmer is strongly encouraged to use the associated object-oriented programming language, generally Smalltalk or C++.[17] However, the vast majority of programmers today (91 percent according to a 1994 survey[18]) have traditional skills and do not yet know object technology or its associated languages. Because almost all interesting applications today still require that logic be written, the requirement to use an object-oriented language significantly curtails the adoption of this class of construction from parts tools by a large percentage of programmers. Thus, the power and benefits associated with construction from parts and object technology are not yet being used by most programmers.

To remedy this situation, techniques have been developed to bridge between the construction from parts world and the traditional world. One approach is to attach construction from parts facilities to traditional languages. This is an initially attractive solution because the WYSIWYG and visual linking metaphor becomes accessible to traditionally skilled programmers, thus boosting their productivity. However, the productivity increase is in part attributable to the granularity and availability of parts. Because most object technology environments emphasize construction from parts, probably more parts will be available for the Smalltalk or C++ developer than for the COBOL or FORTRAN developer. Another drawback is that this approach typically can support *event-to-action*[19] connections, but it cannot handle some forms of the *attribute-to-attribute*[20] connections without changes to the existing language environment. As a result, this second category of visual links, which provides data synchronization, is generally omitted. For these reasons, other approaches must be explored.

Another approach, advocated by many in the construction from parts community, uses *wrappers*. Wrapping allows surrogate parts for the legacy or nonnative code (i.e., something other than the object-oriented language upon which the IDE is based) to be utilized within the IDE. In some cases, this wrapping process can be partly automated—if the IDE has facilities that support the language and constructs of the legacy code fragment. Tools providing this support parse the legacy code and create a public interface for the surrogate part, providing access to the data and services embedded in the legacy code. For example, the wrapper for a C-language dynamic link library (DLL) will contain *actions* for each of the public entry points. This technique is quite useful for facilitating black-box reuse.[21] All of the object technology capabilities of the IDE are still available—all forms of linking are supported and true object-oriented parts can be used—thus future object-oriented enhancements should be possible for an application developed in this way. The drawback to this approach is most apparent for white-box reuse[21] or when the solution requires that new nonnative code be written. Because the wrapped parts are developed in another language and IDE, the develop-

ment process can become disjoint. Not only is the specification of the various portions of the solution split between environments, but so is the debugging. In other words, the logic and data portions of these nonnative parts are not treated as first-class parts (FCPs).[22] This reduces the productivity gains that would ordinarily be achieved within construction from parts environments.

While the above techniques have merit, we propose two companion strategies that may more effectively bridge the technology and skills gap between traditional and object technologies. To expand the usefulness of traditional artifacts within the construction from parts framework, logic and data constructs in a traditional language can be promoted to be first-class parts. Likewise, to enhance the power of traditional languages, part and object scripting can be added.

Two strategies may more effectively bridge the skills gap between traditional and object technologies.

Traditional artifacts as first-class parts. This approach is effectively the reverse of attaching construction from parts frameworks to traditional languages, at least in terms of perspective, and has many characteristics in common with the wrapping technique. Rather than attaching construction from parts capability to existing traditional development environments, the existing language constructs (and their associated development facilities) can be made accessible and manipulable from within the construction from parts framework. In other words, the logic and data constructs in a traditional language can be turned into first-class parts. As an FCP, each of these logic and data parts must define a public interface to its actions, attributes, and events.

In addition, edit, debug, and build capability for these logic and data parts must be provided within the context of the construction from parts framework. This ensures that all aspects of creating a GUI client/server transitional application, using both visual programming and traditional techniques, are integrated as seamlessly as possible.

To illustrate the technique of promoting traditional artifacts to be first-class parts, we will describe the architecture within VisualAge Generator, which includes the same construction from parts framework as VisualAge for Smalltalk.

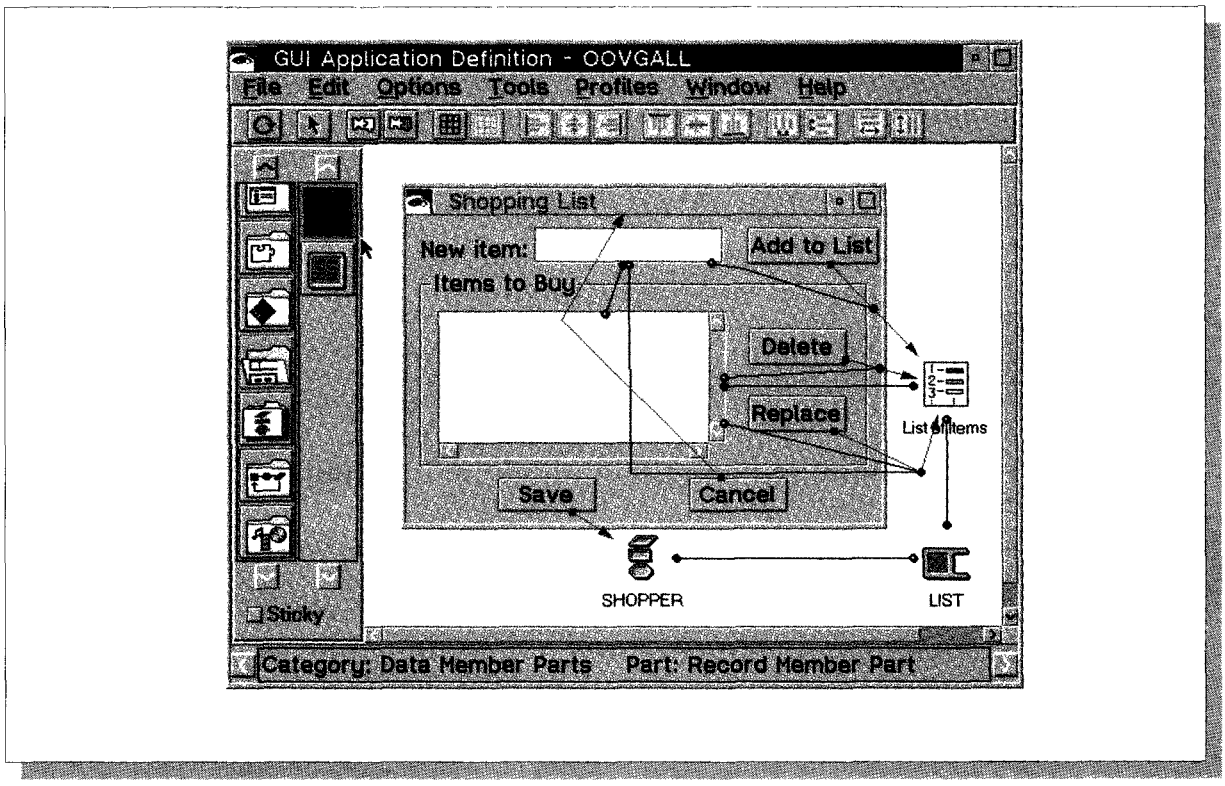
Adding parts palette entries. The first step toward making an entity a first-class part is to make it accessible from within the construction from parts framework. To do this, in VisualAge Generator we added two new categories to the palette of available parts: *data member parts* and *logic member parts*. Within the data-member-parts category, there are two parts: *record member part* and *table member part*, shown in Figure 1. Similarly, within the logic-member-parts category, there are four parts that represent the 4GL logic-entity types.

The data and logic parts are automatically accessible within the construction from parts framework by virtue of being present in the 4GL library (see Figure 2). The public interface of the part is automatically derived from its library definition. The data and logic parts can be selected by name, dropped on the free-form surface, and connected to other visual and nonvisual parts. Part editors can be invoked directly from the visual programming tool. A detailed example of how traditional data and logic parts are used in the construction from parts framework follows.

The example used throughout this section is the shopping list application system shown in Figure 1. It is a simple system consisting of a GUI and a server application called SHOPPER. The GUI, in the box labeled "Shopping List," contains fields with descriptive labels. The central objects in this application are the entry field labeled "New item," the list within the group box labeled "Items to Buy," the nonvisual part labeled "List of Items," the server application labeled "SHOPPER," and its only parameter labeled "LIST." LIST has two fields (not shown): CUSTOMER and GROCERY-ITEMS. The nonvisual part List of Items is a Smalltalk object instantiated from the OrderedCollection class. It serves as the "model" object[23] for the items in the viewable list.

Each shopping item is added to the list by entering it into the "New item" field and then clicking on the "Add to List" button. This occurs because the *clicked* event defined for the "Add to List" button is connected to the *add*: action defined for the "List of

Figure 1 Data Member Parts category with the Record Member Part icon selected



Items" part. The parameter is filled by connecting the *object* attribute defined for the "New item" field to the link itself. List of Items remains synchronized with the items in the Items to Buy box because its *self* attribute is connected to the *items* attribute of Items to Buy. Similarly, GROCERY-ITEMS also remains synchronized because it too is connected to the *self* attribute of List of Items. To fill the parameter to the server application, the *LIST* attribute of SHOPPER is connected to the *data* attribute of LIST. Finally, to cause the server application to run, the *clicked* event of the "Save" button is connected to the *execute* event of SHOPPER.

The remaining connections are not relevant to this paper and are not described.

Defining the public interface. The next step in creating a first-class part is to define the *public interface* for each of the new parts. To build this public or part interface, the following must be defined:

- The *actions* that the part can provide to other connected parts and scripts. These actions are the services that this part can perform.
- The *attributes* or data that the part contains. These include the ways in which the data can be accessed.
- The *events* or messages that the part will signal and send to other connected parts. These events can be used to trigger actions on other parts.

While the public interface for typical object-oriented parts can be extensive, for traditional parts it is often quite simple. An example of a logic-oriented construct, the part interface for a VisualAge Generator application member part, is shown in Figure 3. We next explain three of the elements of this interface for the SHOPPER application member part.

One of the most important actions is *execute*. When triggered, this action will cause the 4GL logic in the associated application member to execute or run, just as if it were invoked from the command line or from another piece of logic.

Figure 2 Traditional artifacts as first-class parts. Logic and data entities in the 4GL library are accessible as first-class parts within the construction from parts environment.

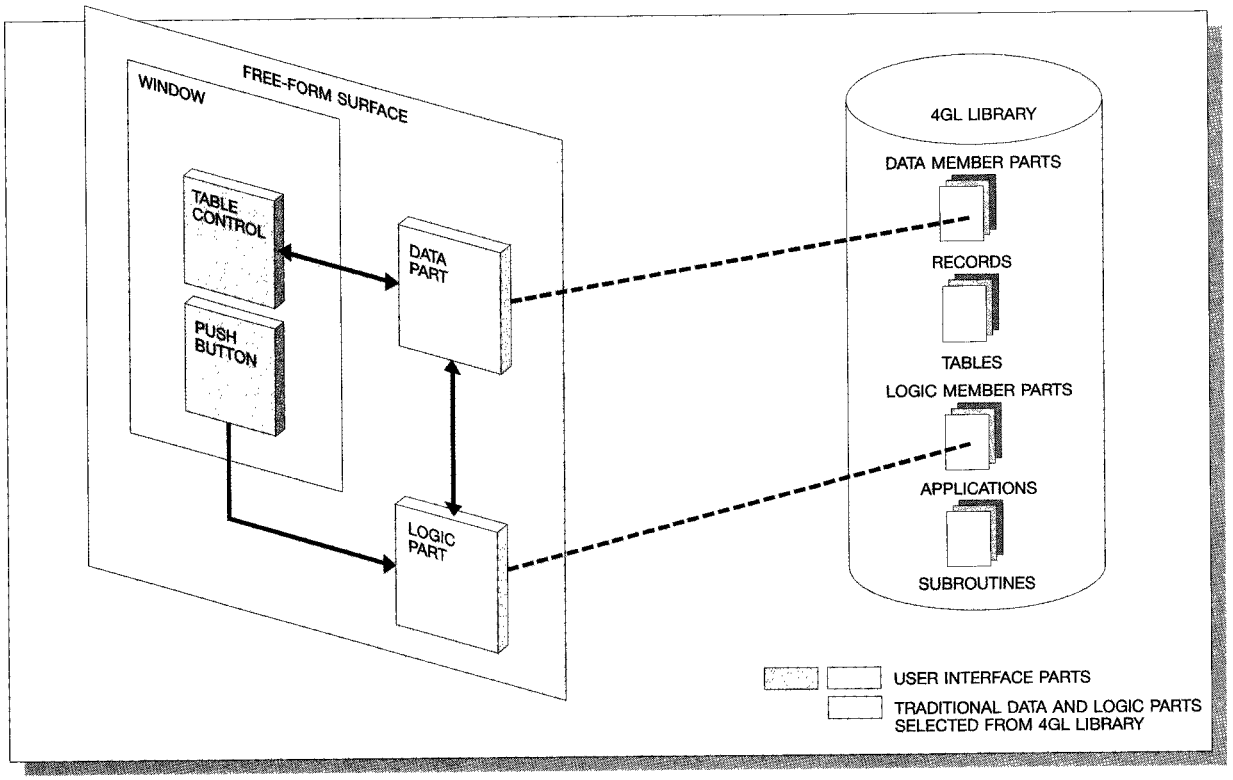
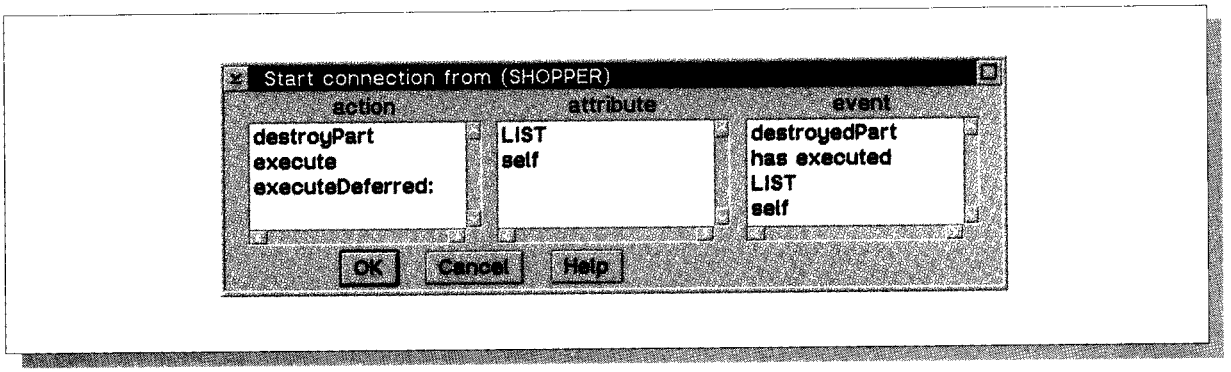


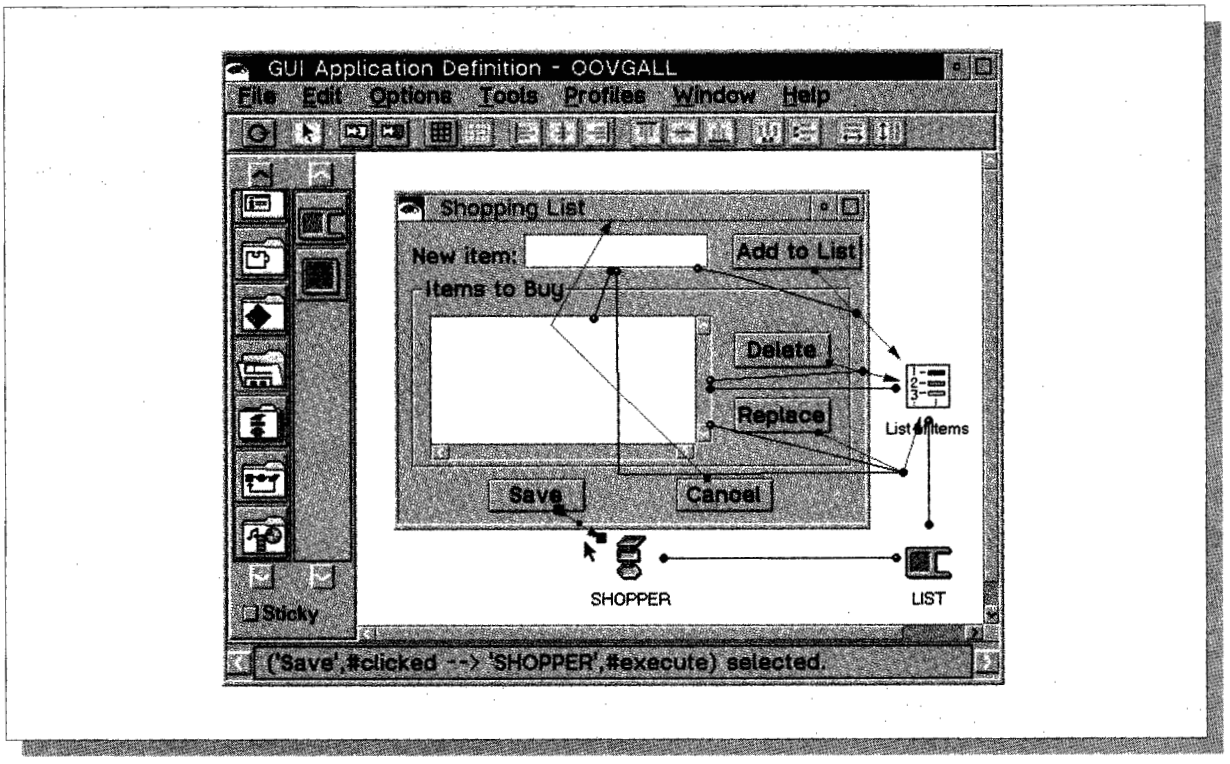
Figure 3 Public interface for a logic part, the application member part SHOPPER



One of the attributes is *LIST*. This record is the only parameter passed to the application. To run the application, the value of this attribute must be provided from some record member part defined on the free-form surface. Note that an application member part may have zero or more attributes that represent its parameters. The attribute *self*, which represents the application member part itself, is the only attribute that is present in all cases.

An important event is *has executed*. This event is signaled when the execution of the associated appli-

Figure 4 Example of visually linked logic. The selected visual link will trigger when the Save push button is clicked. When this occurs, it will cause the SHOPPER application member part to run.



cation member is complete. It allows other actions or attributes to be manipulated at the completion of the 4GL logic, such as after a server application has retrieved a customer record.

With the set of actions, attributes, and events listed in Figure 3, the server application SHOPPER can be fully utilized within the construction from parts environment. Thus, rather than burying the call to SHOPPER within some procedural client code as follows (in pseudocode):

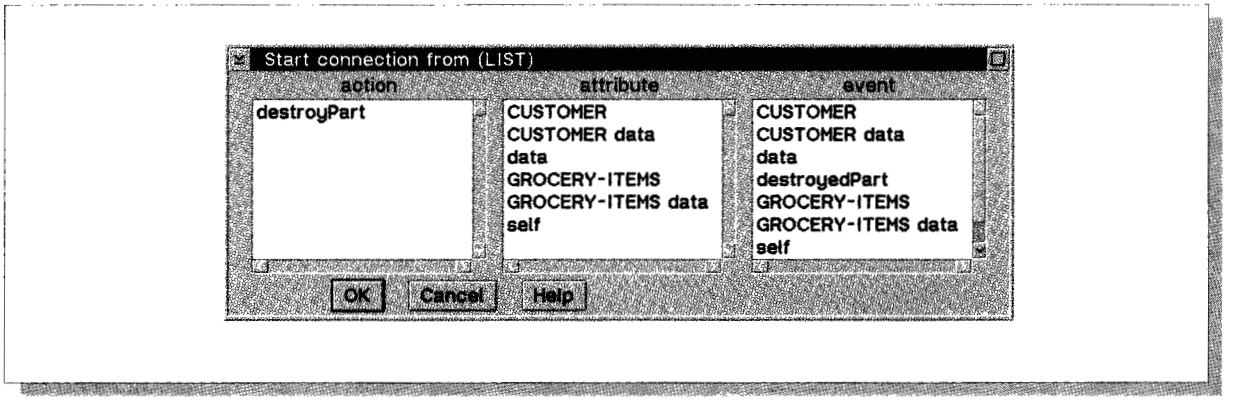
```
switch (UI event):
   Add-to-List clicked:
      add item from entry field to list
   Delete clicked:
      delete selected item from list
   Replace clicked:
      replace selected item with item in entry field
   Cancel clicked:
      close window
   Save clicked:
      CALL SHOPPER (LIST)
```

the invocation of SHOPPER is tied visually to the Save push button as illustrated in Figure 4 and described earlier. The unidirectional arrow from the Save push button to the SHOPPER application member part makes the relationship very clear.

Traditional parts include data-oriented parts as well as logic parts. As described earlier, LIST is the record member part in our example. Figure 5 shows its full public interface.

The primary purpose of data-oriented parts is to hold information. Therefore, there are no actions of particular interest that a record member part provides. On the other hand, there are many attributes available, corresponding to the fields defined within the associated record member. For example, because the LIST record member has a field named CUSTOMER, the associated record member part has two attributes: CUSTOMER and CUSTOMER data. The former is the actual Smalltalk object associated with the CUSTOMER field. It is used when both the field's

Figure 5 Example public interface for a data part. This is the public interface for the record member part LIST.



meta-data, such as the type and length of the field, and its value are required. The latter is used when only the value is needed.

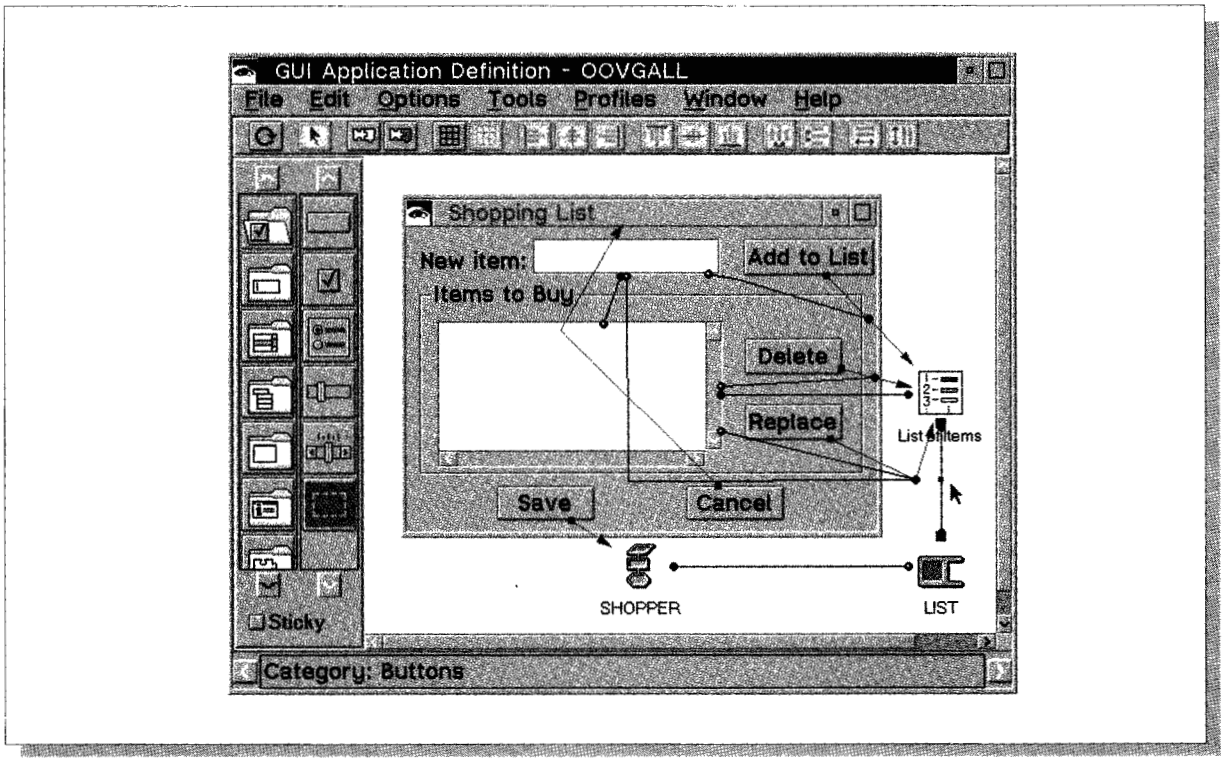
Changes to the value of a data-oriented part cause events to be signaled, which can then cause other actions to occur (due to connections). Within LIST, the *CUSTOMER* event will signal whenever the value associated with the CUSTOMER field is altered. Similarly, the *CUSTOMER data* event will signal whenever the data associated with the CUSTOMER field *or any of its subfields* are altered.[24]

With the set of attributes and events listed in Figure 5, programmers with traditional skills can use what they already know about their data within the construction from parts framework. For example, to keep the List-of-Items collection synchronized with the GROCERY-ITEMS array within LIST, the developer needs only one connection, as explained earlier and illustrated in Figure 6. This eliminates the need to write *any* application code for data movement or data synchronization between these two entities. Surely this single visual link is more expressive than some number of lines of code, probably distributed throughout the application.

Although easily done in VisualAge Generator, it might be nontrivial for some traditional languages to provide support for the data events. The semantics of the attribute-based events, such as *CUSTOMER data*, are that the event will be signaled when the attribute changes. This change can occur either visually (e.g., via an attribute-to-attribute connection) or via logic (e.g., MOVE "Sally" to LIST.CUSTOMER). Because the attribute-to-attribute connection is handled within the construction from parts framework, it can easily be intercepted, thus allowing the target attribute's associated event to signal. However, if the data are updated via traditional logic, this is *outside* the construction from parts framework. Some mechanism must then be established to capture data changes and then cause the appropriate signaling to occur within the parts world. Within VisualAge Generator, this was readily achieved because the 4GL constructs are implemented in a Smalltalk object model on the client, thus allowing the model to be augmented to notify the construction from parts framework when data changes occur. Within a language such as COBOL, which currently has no facility for detecting data changes within logic, enhancements would be needed. While the implementation must perform adequately, the ability to signal data changes could utilize a mechanism similar to that used for monitoring watchpoints (monitoring data states while the program runs).

Providing browsers and debuggers. The next step in making traditional artifacts into first-class parts is to ensure that manipulation of the code is integrated seamlessly within the construction from parts framework. Thus, the new parts must be editable directly from the free-form surface. In addition to the public interface, an editor must be registered with each new part. Within the VisualAge for Smalltalk construction from parts framework, this registration requires that an alternate implementation for the "edit has been requested" message be provided. To edit traditional parts, the definition facilities and editors native to the language of the part are exploited.

Figure 6 Example of visually linked data. The selected visual link will cause GROCERY-ITEMS (within LIST) to remain synchronized with the contents of the list box within the Items to Buy group box.



After an application is defined, it rarely runs successfully on the first attempt. In order to debug this hybrid form of application efficiently, the construction from parts framework must be augmented either to display the traditional logic within its debugger or to invoke the debugger from the traditional language whenever control passes to a traditional logic part. This ensures that program errors associated with the interaction between the object-oriented world and the procedural world can be isolated quickly. In comparison, black box or separated testing suffers with respect to iteration speed and productivity.

In VisualAge Generator, the logic and data parts are edited using specialized definition facilities, with selection based upon the type of the member. For example, when an application member part is selected for editing, the graphical Application Definition facility is opened. Similarly, when the debugging activity crosses into the 4GL code, the Interactive Test facility within VisualAge Generator is activated. In the future, this support could be enhanced by choosing either the Smalltalk debugger or the Interactive Test Facility as the preferred interface, and then displaying both the Smalltalk and the 4GL logic and data within a single facility.

Integrating the build process. While for some implementations the features just described might be sufficient to claim first-class part status for traditional artifacts, more can be done. Once an application is defined, an executable module must be built. This involves several steps—code generation, transfer of code to the target platform, compilation, and linking. It is desirable for this process to be handled by the development environment, leaving to the user only the initiation of the process. Further, the developer of transitional applications should not be required to initiate multiple builds just because both object-oriented and traditional technologies are exploited.

Seamless build processing can be supported in at least two ways. One way is to make the traditional parts themselves responsible for initiating a tradi-

tional build when they are asked to do so from within the construction from parts environment. Another way is to have a configuration management facility that understands both object-oriented and traditional constructs. It can then be responsible for building all portions of the applications.

The second option implies that the application components are managed by, or at least known by, a single facility. Because there are activities other than build that require the developer to interact with the underlying library, this is the final aspect of integrating the traditional and object world. When all components of an application can be archived, queried, etc., within a single tool, the productivity of the developer will increase.

In summary, the benefits of making traditional components into first-class parts are that these components are directly usable within the construction from parts framework, allowing developers with traditional skills to quickly exploit the visual linking metaphor without having to invest the time to learn a new object-oriented language. Additionally, when modifying these traditional components, or when creating new ones, the facilities that the developer is familiar with—the editors and the debuggers—are immediately available. Finally, when packaging the entire application for distribution, having a single facility responsible for the process further integrates the two technologies and boosts productivity. The result is that programmers can remain productive while they learn object technology at their own pace.

Part and object scripting from within traditional languages. The previous section described how traditional language constructs can be manipulated within the object world. While this is quite important, it is only one side of the bridge between object and traditional programming. When operating within the traditional world, it is equally desirable to have access to the parts on the object side. This section describes an approach to providing this support.

While visual programming is quite powerful, many operations are more easily implemented and more easily understood by others when coded directly. Furthermore, situations that involve conditions and loops rarely can be done visually, requiring that code be written. For these reasons, all visual programming IDEs provide some form of scripting or implementation language to supplement the visual tools. Unfortunately for many developers with traditional skills, the language is often an object-oriented one, such as Smalltalk or C++. Therefore, to facilitate use of visual programming environments by these developers, traditional languages can be enhanced to provide access to and manipulation of the visual and nonvisual parts. This does not mean that these traditional languages must become fully object oriented; rather, they only need to be able to access, or get addressability to, parts in the visual programming name space, and then to be able to send messages to them. In other words, part and object scripting needs to be supported.

Again we will use VisualAge Generator to demonstrate this technique. There is a name space for the visual portion of each GUI application and another one for the 4GL logic associated with it (see Figure 7). Because the 4GL does not currently know about parts, constructs are being added to support part and object scripting. This will be accomplished with additions similar to the following:

- Addressability. *ObjectHandle* is added as a new type of variable. Then, two built-in functions are added to return an object handle for a named entity. *EZEPart(name)* is added to return a handle to the part (in the construction from parts name space) identified by the *name* parameter. This part can be a visual or a nonvisual part. *EZEObject(name)* is added to return a handle to the named Smalltalk object. This object can be any global variable within the IBM Smalltalk system, including classes. An example from Figure 8 is:

  ```
  save_pb = EZEPart ("Save");
  ```

  This line of code assigns the object handle of the Save push button to the variable save_pb.

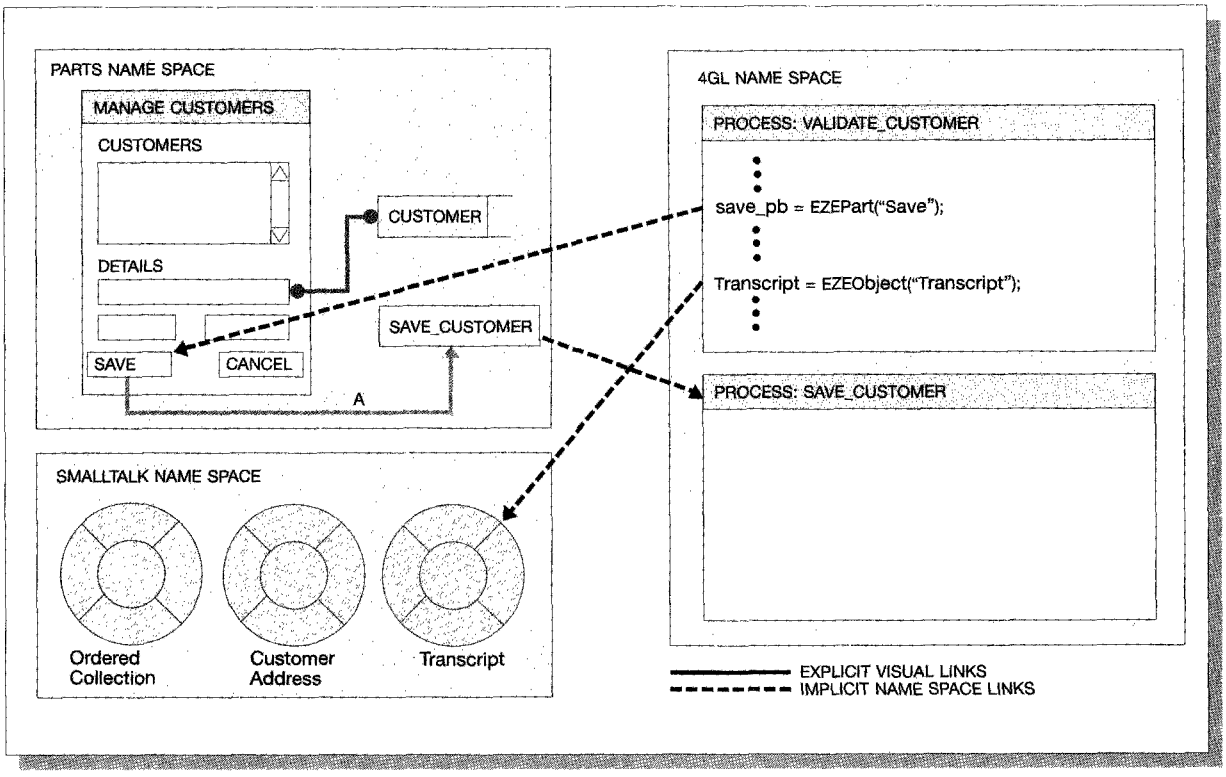
- Messaging. The built-in function *EZESend(receiver, selector, arg, . . .)* is added to allow manipulation of parts and other named entities. This includes the ability to get and set part attributes. An example from Figure 8 is:

  ```
  EZESend (save_pb, save_message);
  ```

  In the example, save_message is set to either "enable" or "disable." This function call will cause the desired enablement state to be set.

- Signaling. *EZEPartSignal(part, eventName)*, a built-in function, is added to allow events within the construction from parts framework to be signaled. An example usage might be to indicate that a given process has finished executing:

Figure 7 **Bridging between separate name spaces. The parts, objects, and 4GL name spaces can be bridged as shown. In the parts-to-4GL case, visual connections, such as "A," cause part wrappers to invoke services on their underlying implementations, such as the 4GL process SAVE_CUSTOMER. In the 4GL-to-parts case, the special functions EZEPart and EZEObject provide accessibility to entities in the object domain.**




```
EZEPartSignal (EZEPart("Validate_ID"), "has
                                executed");
```

Other parts can then register interest in when the function Validate_ID *has executed.*

By adding these three capabilities, the VisualAge Generator language will be extended to support full access and manipulation of the parts, classes, and objects within both the construction from parts framework and the underlying IBM Smalltalk system. Adding this support is straightforward in VisualAge Generator because Smalltalk is the underlying language in both name spaces; however, this technique can apply equally well to other languages, such as COBOL.

Enhancing a traditional language to provide these capabilities does not turn the traditional language into an object-oriented language, but it opens the language to some of the power and flexibility available within object-oriented languages. As enterprises increase their object technology skills, the parts and classes created can be exploited immediately by all developers. In this way, businesses can decide to develop object technology skills at their own pace without forfeiting the benefits in the near term. This technique offers each developer two options for scripting: the native object-oriented one and a traditional one. Usage can then be determined by the skills and comfort level of each developer. In some enterprises, developers will make the transition rapidly to native object technology; in others the change will occur more slowly. In either case, the benefits of object technology and visual programming can be realized immediately.

Rapidly developing transitional applications

The previous section described two techniques for bridging between the procedural and object-oriented worlds. While this programming-level support can

greatly ease the creation of transitional applications, additional technologies can be exploited to allow such applications to be built even more quickly and thus further ease the transition to object technology.

Rapid application development (RAD) is a term used to describe methodologies and tools that claim to deliver higher application creation and maintenance productivity than typical 3GLs and their associated tools.[25] This enhanced productivity may be attributable to a higher level of specification (e.g., 4GLs, CASE [computer-assisted software engineering] tools, and templates), greater portability across platforms (e.g., interpreters, code generators, portable class libraries and frameworks), tighter integration of tools (e.g., IDEs), or a combination of these. While the techniques employed to raise productivity vary, a central theme can be found: each approach attempts to reduce both the conceptual and the manual work required of the application developer.

Within the domain of object technology, many of the same techniques are being utilized. For example, visual programming attempts to raise the level of specification by replacing user interface code with visual links. Class libraries, such as those in VisualAge for Smalltalk, provide portability of code across many different platforms. However, most of the RAD options that support object technology have little support for integrating the traditional portions of transitional applications.

In the section that follows, we introduce data-oriented, template-based generation as an example of rapid application development within the context of transitional applications. Since the technique is reasonably well known from its use within the procedural world, we focus on its applicability to transitional applications. In particular, we discuss how template technology can significantly "jump start" the creation of all aspects of transitional applications, including the object-oriented GUI, visual links, client logic, and server database access logic.

Rapid application development using template technology. One of the rapid application development methods especially useful for the transition to object technology is data-driven template technology. Templates have been used for several years in the procedural domain for the automated construction of user interface, client and server logic, and data access code that performs standard operations on database information. Extending this to include the creation of user interface, logic, and data access parts within the construction from parts framework can help mask the object vs procedural boundaries. Thus, this technique can help make visual programming and object technology even more approachable by enterprise developers.

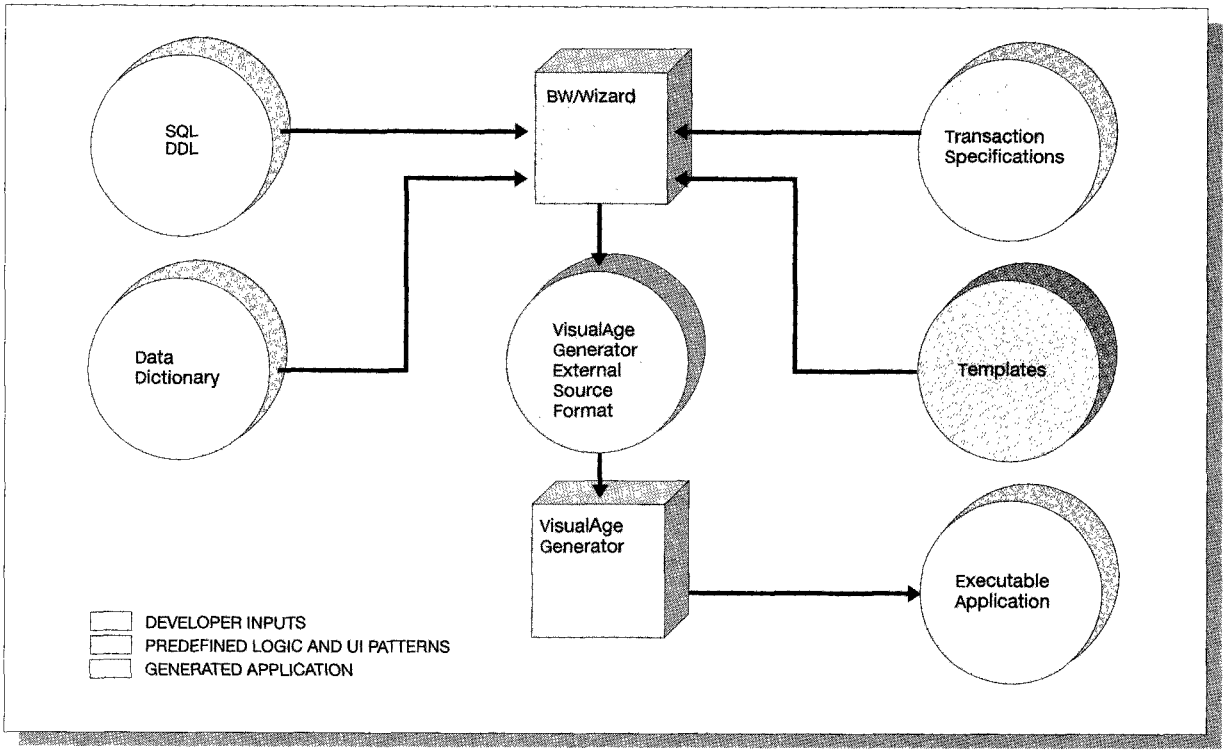
Figure 8 **Part manipulation within VisualAge Generator. This example shows how parts within the construction from parts framework can be manipulated easily within 4GL logic by using a small set of built-in functions. In this sample code fragment, the part named "Save" is either enabled or disabled, depending upon the result from the function Validate_LIST. This code could be used with the example in Figure 6 to enable Save only at the time that one or more groceries are added to the list.**

```
save_pb = EZEPart("Save");

if (validate_LIST( ) = 0);
   save_message = "enable";
else;
   save_message = "disable";
end;

EZESend(save_pb, save_message);
```

Business applications are good candidates for RAD template development, since the logic of many business applications involves entering, storing, retrieving, and modifying records stored as rows in relational database tables. A table row can be represented within the construction from parts framework as a part where each column represents a unique attribute. Select, insert, update, and delete are standard actions for a table-related part. The logic associated with these methods differs only in the specifics of the table and column definitions.

Implementing templates for standard business applications within the construction from parts environment requires identifying, and then abstracting, the common aspects of the user interface, logic (e.g., field validation), and data access, creating skeleton or template parts. This abstraction process must ensure that the template is independent of particular table and column definitions. The template then enables the generator, given a particular table and column definition, to automatically produce a default GUI with connections to nonvisual logic and data ac-

Figure 9 Template-driven development



cess parts (see Figure 9). Thus, the essential but mundane aspects of a graphical, client/server, data access application can be built simply by providing table and column definitions. This frees the enterprise developer to focus time and energy on user-interface customization, unique business logic specification, and system tuning.

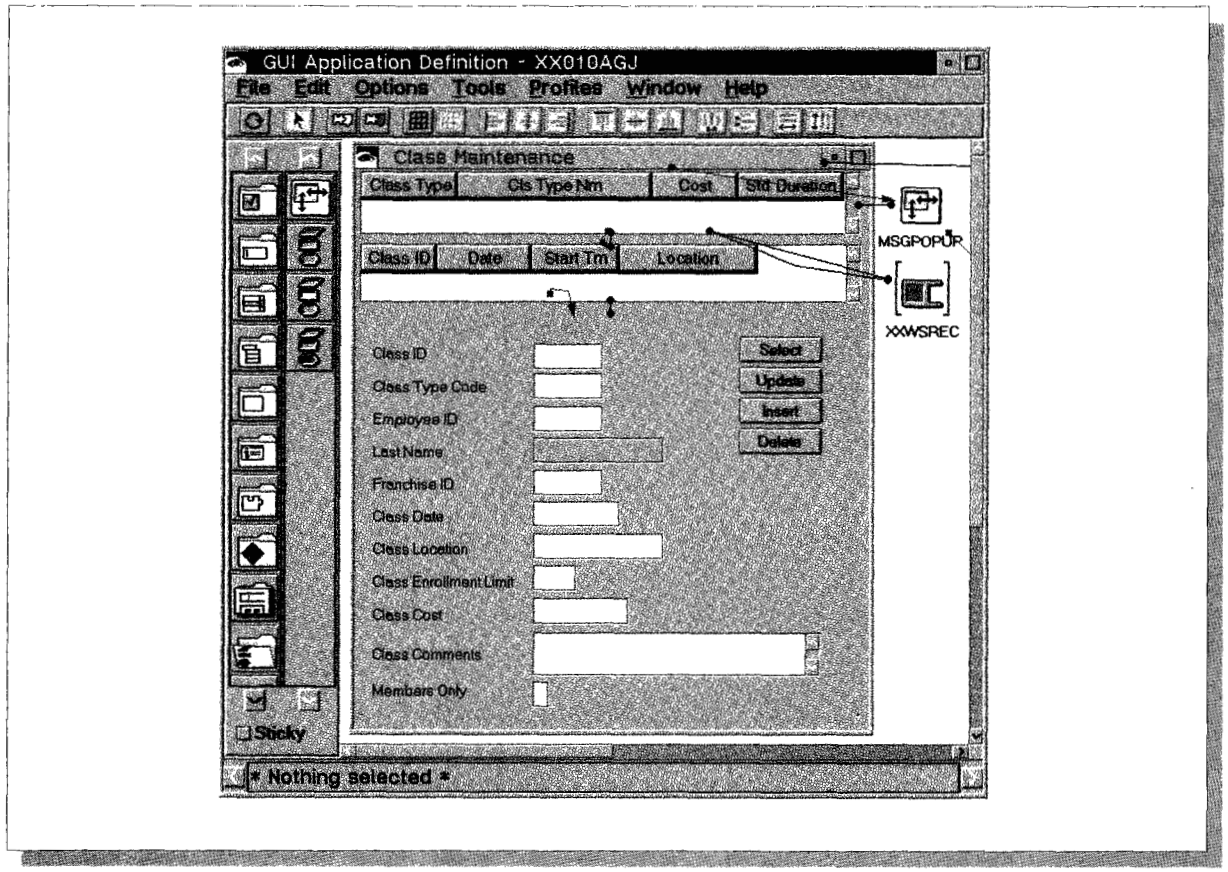
Figure 10 shows an example of an embeddable GUI part ("Class Details") embedded in a class maintenance GUI application. This part was built automatically using RAD template technology[9] that provides input to VisualAge Generator. The input to the RAD generator was a database table representing the classes being taught in a health club. The public interface for the class includes the actions *executeSelect*, *executeInsert*, *executeUpdate*, and *executeDelete*. Entry fields in the window are used for the display and update of attributes for the currently selected class.

The relatively simple public interface for the Class Details part hides a fairly complex GUI part definition (see Figure 11). The definition includes many connections to first-class parts that represent traditional artifacts. Among other things, these parts include remote procedure calls to a server application that accesses the database.

As illustrated in the two examples, data-driven template technology can create a significant, essential portion of a typical business application with very little effort and, equally important, with very little object technology skill. The enterprise developer creates the embeddable GUI part, the traditional parts, and the associated server applications by providing the relational database table definition, then requests generation of a row maintenance part for the table.

The automatic creation of database access parts, default GUIs, and separate client and server logic eases the transition to an object technology environment in several ways. The RAD templates or application models can embody the knowledge of experienced developers. By reusing this knowledge, template-based generation can automatically provide many

Figure 10 Use of a template-generated GUI. While the embedded GUI part Class Details is rather complex, its use within the Class Maintenance application is very simple.



features of a well-engineered transitional application, such as:

- Simple object interfaces to complex transaction processing
- Separation of client and server so that data flow across the network is minimized
- Database access with availability (no locking during user think time) and integrity (prevention of simultaneous updates)
- Data validation, formatting, and referential integrity checking built into the user interface
- Consistent exception handling

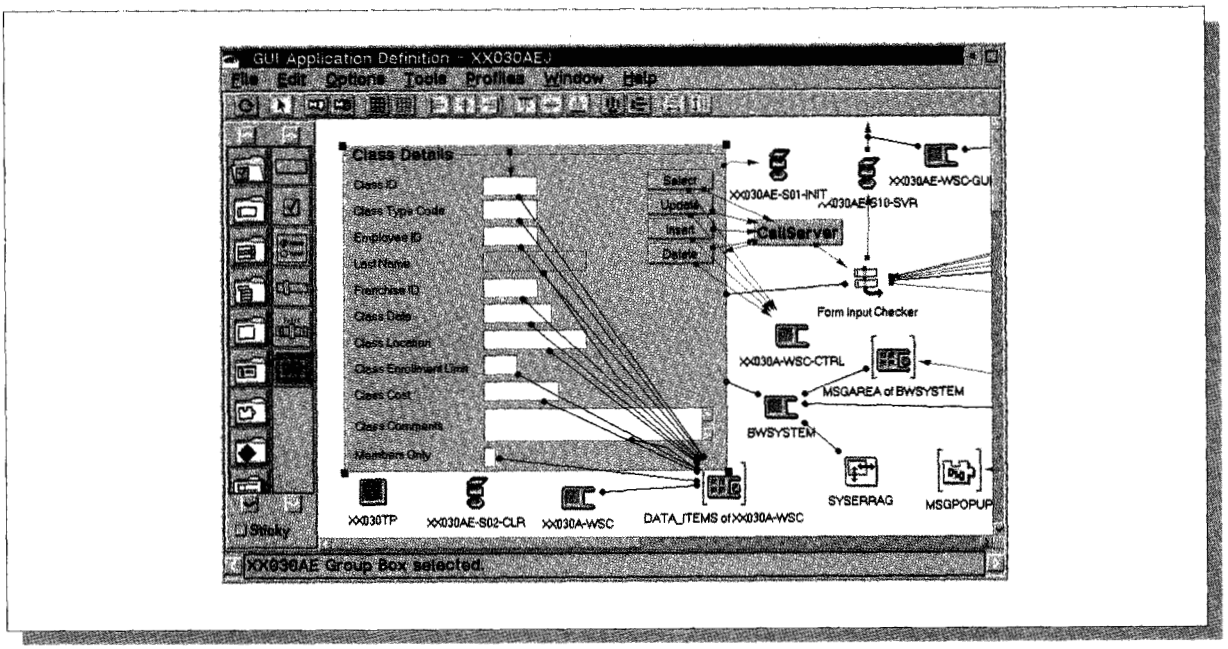
Additionally, parts built using RAD template technology can improve maintainability. For example, all template-generated applications use a common architecture, faithfully adhere to enterprise standards (which can be built into the templates rather than being left up to developers), and present the same "look and feel," both to the end user and to the developer.

In summary, as more and more enterprise developers with traditional skills move to object technology, rapid application development techniques, such as template-based code generation, should be considered. These RAD techniques have proven to be valuable to many of these same developers for procedural software. This familiarity may further accelerate the rate at which these developers create the transitional applications that bridge between, and yet still exploit, the traditional and object paradigms.

Conclusion

Object technology is a powerful software advance that can be effectively used today by enterprise de-

Figure 11 **Embedded GUI part generated using templates. This rather complex embeddable GUI part was created automatically (aside from some repositioning of fields) by a data-driven template generator.**



velopers to build transitional applications. This can be accomplished by using tools that bridge between the object technology and the traditional worlds, and that hide much of the associated complexity. Utilizing these tools and techniques, transitional applications can be built that provide the strengths of visual programming and object technology, yet remain approachable by most enterprise developers. In particular, by introducing traditional parts into the construction from parts framework, and by adding part and object scripting to traditional languages, the exploitation of object technology by developers with traditional skills becomes easier. Rapid application development techniques, including data-driven template-based code generation, can be exploited to raise productivity and mask complexity, thus further enhancing the integration of the two technologies.

While we have presented several strategies for allowing the power of object technology to be exploited by a wider audience, many more techniques exist and should be considered. Furthermore, if object technology continues to grow in popularity, strategies of this sort will become even more crucial in order to tap the skills, experience, and resources of the world's vast number of traditionally skilled programmers.

Acknowledgments

The authors would like to thank Jeri Petersen, Yen-Ping Shan, and Michael Wheatley for their very thorough reviews and critiques of earlier drafts. We also appreciate the contributions of Peter Brennan and Peter J. Sperling of Bridgewater Consultants. Finally, we especially acknowledge all of the developers, past and present, in the VisualAge family organizations within IBM and those within Bridgewater Consultants who have incorporated many of these ideas into real products.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Bridgewater Consultants, Inc. or Sun Microsystems, Inc.

Cited references and notes

1. P. Wegner, "Concepts and Paradigms of Object-Oriented Programming," *OOPS Messenger* **1**, No. 1, 7–87 (August 1990). Wegner provides a thorough discussion of the many aspects of object-oriented programming.
2. D. A. Taylor, *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley Publishing Company, Reading, MA (1992).
3. *VisualAge for Smalltalk: User's Reference*, SC34-4519-00, IBM Corporation (1994); available through IBM branch offices.

4. The VisualAge family includes VisualAge for Smalltalk, VisualAge Generator, VisualAge for C++, VisualAge for COBOL, and VisualAge for Basic.
5. As used here, the term *scripting* refers to the ability to access and manipulate visual and nonvisual parts. Further, our use of the term includes the ability to access and manipulate objects and classes in the underlying object-oriented language that supports the construction from parts framework.
6. M. Goetz, "Forum: A 4GL Future?" *Software Magazine* **16**, No. 3 (March 1996).
7. The term *transitional* is used here as it is in the context of home styles in the United States, where traditional, transitional, and contemporary represent three distinct building styles. Rather than representing a temporary state, a transitional style includes elements from both traditional (e.g., materials) and contemporary (e.g., open living spaces) styles. In a similar manner, transitional applications combine characteristics of both traditional (e.g., language constructs) and object-oriented (e.g., parts, messages) technologies.
8. VisualAge Generator Version 2.2 (known in previous releases as VisualGen) is an integrated development environment (IDE) that incorporates the *traditional part* and the *part and object scripting capabilities* described in this paper. VisualAge Generator supports the development, test, and generation of client/server applications for a wide variety of run-time environments.
   GUI client user interfaces are defined via visual programming using the VisualAge for Smalltalk construction from parts framework. Client logic and server programs are defined using a procedural 4GL. The 4GL includes simple, predefined verbs for file and database access and user interface presentation, and it supports datastore-independent data entities and general-purpose logic constructs. The 4GL logic and data entities are wrapped by the visual builder, automatically creating the *traditional parts* within the construction from parts framework described in the paper.
   The visual programming, 4GL definition, and source language test and debugger operate as a single, integrated development environment. The developer can move back and forth between the tools during iterative application development. A notification framework ensures that all tools remain synchronized.
   The 4GL has statements for conditional logic and for assigning values and mathematical expressions to variables, calling or transferring to other programs, and finding and retrieving values from tables of information. Verbs for accessing persistent data allow the developer to insert, update, delete, and retrieve records from a variety of datastores (relational and hierarchical databases; and indexed, relative, and serial files). For further information, refer to *Introducing VisualAge Generator Version 2.2*, GH23-0225-00, IBM Corporation (1996); available through IBM branch offices.
9. The BW/Wizard template-driven generator was used to create the row maintenance part and the related traditional parts and server programs used in our example of exploiting *rapid application development templates* for automatic generation of transitional applications.
   Templates are reusable sequences of source code. Early examples of template technology include assembler and high-level language macros. C++ member functions and class templates are another example.
   BW/Wizard templates model entire applications. Traditional parts are modeled in VisualAge Generator External Source Format statements; GUI parts are modeled in IBM Smalltalk statements in fileOut format. The template control language surrounding the model source statements supports a multilevel scheme for symbol substitution that allows the same code sequence to be repeated multiple times, once for each column in the relational database table. A data element dictionary provides information for columns that is not captured from the database table definition. The additional information includes user interface labels, descriptions, and formatting and validation requirements, which are incorporated into the generated parts.
   Template definition is a complex task requiring the attention of skilled programmers who can handle multiple levels of abstraction. Fortunately, this complexity does not confront the developers who use the templates for building applications and who refine the generated outputs with the language editor or visual programming tool appropriate for the generated output. They have no need to understand the template language or its use.
   For further information on BW/Wizard, contact Bridgewater Consultants, Inc., 14168 Poway Road, #201, Poway, CA 92064.
10. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall International Ltd., Hertfordshire, UK (1988).
11. O.-J. Dahl, B. Myrhaug, and K. Nygaard, *(Simula 67) Common Base Language*, Publication N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, October 1970, revised February 1984.
12. *Smalltalk-72 Instruction Manual*, A. Goldberg and A. Kay, Editors, Technical Report SSL-76-6, Xerox Palo Alto Research Center, Palo Alto, CA (March 1976).
13. D. A. Taylor, *Business Engineering with Object Technology*, John Wiley & Sons, Inc., New York (1995).
14. *Visual Object-Oriented Programming: Concepts and Environments*, M. M. Burnett, A. Goldberg, and T. G. Lewis, Editors, Manning Publications Co., Greenwich, CT (1995).
15. A. Kessler, "Fire Your Software Programmers—Again," *Forbes ASAP* (August 29, 1994).
16. J. R. Hines, "Program Notes: Visual Programming," *IEEE Spectrum* **32**, No. 11 (November 1995).
17. While one is allowed to write logic in a language other than the provided object-oriented language, the process is far more involved than if the native object-oriented language is used. Debugging across the language boundaries can be particularly unpleasant.
18. C. Jones, "Gaps in the Object-Oriented Paradigm," *IEEE Computer* **28**, No. 3, 70–71 (March 1995).
19. *Event-to-action* connections cause an action to be performed when an event occurs. For example, connecting the #clicked event of the push button to the #openWidget action of the target window causes a window to open when a push button is clicked.
20. *Attribute-to-attribute* connections cause the attributes of two parts to always remain synchronized. For example, connecting the #object attribute of the entry field to the #name attribute of the customer object causes an entry field to always be the same as the customer object's name.
21. "Black box reuse" refers to code reuse that requires no knowledge of internal code—the entity can be successfully reused with knowledge of only the interface to the code. With "white box reuse," knowledge of the internals is necessary to reuse the code effectively.
22. We introduce the term *first-class part* to represent the parts that are accessed and manipulated without burdensome restrictions within the construction from parts framework. By definition, we assume that all native parts (written in the associated language) are first-class parts. Parts created in other

languages or with other tools can become first-class parts by supplementing the part with all the access and manipulation services that the construction from parts framework provides for native parts.

23. A good design practice is to separate a "model" object from its "view." The model object is an integral part of the application and maintains some part of its current state. The view reflects that state to the user of the application. This separation allows the user interface to be changed without change to the underlying application.
24. The design decision for VisualAge Generator was to have two attributes and events per data entity. Another tool might have one attribute and event per data entity. There are two primary issues to consider. First, will the tool user need to access the entity as an object? If so, the meta-data should be available as well as the entity's value. In VisualAge Generator, the object (meta-data and value) are used extensively within the construction from parts framework, while the value is used when connecting to the traditional side. Second, if the language supports substructured data fields, will the tool user want to limit signaling up and down the substructure chain? A decision to limit this signaling led to our design choice in VisualAge Generator.
25. Information about RAD is available on the World Wide Web. We recommend the background information at http://www.dsdm.org/backgrnd.html and "The Underlying Principles: Version 2" at http://www.dsdm.org/method.html.

Accepted for publication July 8, 1996.

A. Hayden Lindsey *IBM Software Solutions Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: hlindsey@raleigh.ibm.com).* Mr. Lindsey joined IBM in 1985 after receiving a B.S. degree in mathematical sciences from the University of North Carolina at Chapel Hill. He has been applying object technology and Smalltalk since 1988, primarily in the areas of application generation, single system image debugging, and performance. Mr. Lindsey is currently a senior technical staff member and an architect for the VisualAge Generator product set.

Paul R. Hoffman *IBM Software Solutions Division, P.O. Box 12195, Research Triangle Park, North Carolina 27709 (electronic mail: phoffman@vnet.ibm.com).* Mr. Hoffman joined IBM in 1969 after receiving a B.A. degree in mathematics sciences from the University of Michigan. He is currently a senior programmer and an architect for the VisualAge Generator product set in the area of rapid application development. His initial IBM experience was in operating system design and development in shared-memory multiprocessor locking and resource utilization, security, and remote job entry. Application experience in delivery of computer-assisted instruction in eight different operating environments led to his current position in VisualAge Generator, which generates applications for a wide variety of operating environments. Prior to his current activities, Mr. Hoffman was a designer for VisualAge Generator in the areas of database access, COBOL generation, and client/server communications.

Reprint Order No. G321-5633.