Directions in Software Development and Maintenance

Ted J. Biggerstaff

September, 1993

Technical Report

MSR-TR-93-16

Microsoft Research

Microsoft Corporation

One Microsoft Way

Redmond, WA 98052

Directions in Software Development and Maintenance

Ted J. Biggerstaff
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
tedb@microsoft.com

Abstract

Development environments are entering a period of dramatic change. A major component of this change is a reorientation toward domain driven development environments including an integration of domain oriented support tools. This reorientation will bring about a decline in the role of conventional programming languages and at the same time, force an evolution toward more abstract programming representations -- more abstract in the sense that most of the implementation details (as we know them today) will be abstracted away. Thus, development is moving farther away from conventional software engineering models and closer to the problem.

Key Words and Phrases: Abstraction, CASE, development environments, domain model, problem oriented languages, program generators, representation, visual programming.

1. Introduction

In a recent paper [1], I described the significant reuse benefits that accrue from forms designers, fourth generation languages and interface development tool kits. These tools provide high levels of reuse through domain oriented (often visual) programming techniques. It was clear to me that mainstream programming, was evolving in this direction. What I failed to realize was just how fast this evolution was occurring and how sophisticated such tools have become in the last year or so.

I was surprised recently when I sat down to use one such system on a personal desktop computer. In a matter of a few hours, I learned enough about the system to produce a simple application with a sophisticated interface that included drop down menus for the application functionality; pop up panels for selecting options; standard interaction panels for reading, writing and printing files; a whole set of control buttons plus functionality for manipulating sets of items; list boxes with scroll bars for displaying the items; and much more. A few short hours more and I could have had a sophisticated relational data base built into my application.

Most of this interface was built by graphically cutting, pasting, grouping and editing the reusable components. This was "clip-art" style programming. The total number of lines of real code that needed I had to write was less than 550 and perhaps 10% of that was written by the system for me. This code provided the functionality for 6 menu items, 11 pop up panels, 50 buttons and data selectors, 2 item lists with scroll bars, 2 text editing boxes and 3 file dialog boxes. It probably would have taken me weeks to build as sophisticated an application from scratch, even if some of the items were available as reusable classes, in the main because of the extra effort needed for integration, coordination and testing.

I was familiar with such tools on workstations. They had always struck me as useful but clumsy and hard to use, mostly because of the difficulty associated with the integration of the output of the tool and the rest of the application. Too much manual integration effort was required. Not so with the more recent tools. The tool I used managed everything for me and when I wanted to test my application, I just hit the run button.

This experience brought home the message that the world of development and maintenance is starting to change very fast. I believe that development environments are entering a period of transition that will change their character dramatically over the next decade. This change is composed of several distinct technology changes:

- **Domain reorientation:** Programming representations and environments are shifting from software engineering theoretic viewpoints to domain oriented viewpoints, thereby allowing greater levels of reuse.
- **Representational abstraction:** Program representations will become more abstract avoiding implementation commitments where ever possible. This will foster increased reuse by

eliminating more and more of the reuse limiting, concrete component details.

- **Declining role of programming languages:** Conventional programming languages will be used in a support role and the representation of programs will include more extra-linguistic details that support the generation of the implementation details.
- **Domain driven integration of tools:** Many of the useful CASE tools of today will be integrated with the development tools, but integrated through domain oriented conceptual models rather than software engineering conceptual models.

The first three changes enhance the programmer's ability to reuse components thereby, improving the efficiency of the development and maintenance process, and the quality of the resulting software. The last change is really an engineering consequence of the domain reorientation.

2. Domain reorientation

2.1. The nature and consequences of the change

There are really two interrelated aspects to domain reorientation: 1) a shift in viewpoint and 2) a change in enabling technology. The shift in viewpoint concerns what concepts and relationships provide the most effective foundation for programming. The change in enabling technology concerns whether one builds programs by constructing linguistic forms or whether one builds programs by graphical construction.

For many years programming and software engineering have depended upon a broad general foundation and consequently, their representation tools are best suited for working with small scale structures, e.g., *integers*, *strings*, *functions*, etc., which provide relatively little programming leverage. That is, like gates in hardware, it takes a lot of them to construct a large application. Thus, conventional programming languages and software engineering theories are well designed for constructing small, intricate, non-standard, hand crafted parts.

However, the representation problem is more than a simple matter of scale or grain size. Programming languages and software engineering representations are organized along a different conceptual dimension or viewpoint from the domains that they deal with. While the framework of programming languages and software engineering is built from highly general entities and relationships (which are the small grained structures mentioned earlier), the framework of the problem domain is built from highly domain specific entities such as *panels*, *icons*, *buttons*, etc. and relationships such as *provides values for*, *enables*, *changes value*, etc. (which are the large grained structures). The difference in the conceptual dimension or viewpoint means that it is a long leap from programming languages and software engineering representations to the representation of problem entities and relationships that we are actually dealing with.

This programming language or software engineering viewpoint leads to a highly operational view of the software. That is, developers tend to focus on data flows between variables, calls between functions, and in general, operations that are close to the operational level of the machines upon which the programs are to run. This operational myopia tends to obscure the domain level operations.

Object oriented systems have improved on this state of affairs by increasing the grain size and hiding many of the inconsequential programming level details, but more importantly, by introducing a view of the software (i.e., the class hierarchy) that begins to deviate from the purely low level, operational point of view. That is, classes begin the to shift toward focusing on problem entities, relationships and operations. But object oriented systems still expose many of the programming details and split the programmer's attention between the problem domain objects and relationships, and all of the programming details that are required to implement them. Even when classes are reused in black-box style reuse, the programmer must be aware of many of the low level operational characteristics of the classes because they show through and can introduce bugs [2]. Thus, object oriented languages are not, in and of themselves, domain oriented although they may enable limited domain orientation. They still fall short of the goal of allowing a programmer to operate strictly in terms of problem domain entities and relationships.

We get a true domain orientation if we shift all of the way to POLs (Problem Oriented Languages), although POLs introduce their own particular difficulties (e.g., domain integration). They also have the same language oriented drawbacks as conventional programming languages, object oriented languages and software engineering tools. Their enabling technology is language based and that makes programming harder than it need be. Forming coordinated sets of abstract linguistic expressions seems intellectually harder than cutting and pasting

domain level, visual components that evoke deep levels of domain intuition.

Furthermore, the one dimensional (string-based) character of conventional text languages (programming or POLs) aggravates the problem. For example, text languages require abstruse nesting of brackets, braces or parentheses just to express simple, direct relationships such as, inclusion.

So clearly, shifting the viewpoint toward domain based objects and relations simplifies the programming job. But if we are also willing to change the enabling technology to a two dimensional form, then there is an opportunity to perform domain oriented programming without many of the compromises that language based enabling technologies promote. This changes the programming job from one of composing abstract linguistic expressions that distantly relate to one's domain objects and relationships to one of assembling graphic parts that directly evoke the concept of the domain objects and relationships.

Thus, the kind of domain reorientation that I envision is one in which both the viewpoint is shifted and the enabling technology is changed. This style of domain reorientation arises when the programming representation scheme is visually tailored to a specific problem domain by mapping the domain entities and their relationships into a visual metaphor that captures those entities and relationships in obvious, natural graphical forms. Each domain entity has an iconic representation that naturally evokes the concept of the domain entity the icon defines (e.g., the use of an icon that looks like a control button to represent an actual control button on a user interface). Similarly, the relationships among the domain classes (e.g., a button that provides services for a dialog panel) are usually mapped into graphical relationships such as graphical inclusion or juxtaposition.

But visual domain reorientation implies more than a passive visual representation of the program. It also generally implies that the development process has become one in which the visual representation of the program is directly manipulated by the programmer to effect development of or change to the evolving program. In a sense, the abstract visual model is the program. The real operational program, to the degree that it is different from the visual model, is typically invisible to the programmer.

In a sense, domain reorientation is saying that developing, maintaining and understanding domain specific applications is better achieved through models expressed in terms of the domain concepts and their interrelationships than by models expressed in terms of programming oriented connections and flows within the program. And in the case where the domain concepts and relationships are represented graphically, the programming process is shifted away from one that is language oriented toward one that is manipulation oriented. That is, the programmer spends less time writing linguistic forms and more time constructing compositions of graphical problem domain objects.

The good news is that domain orientation significantly enhances reuse. This enhancement arises because the narrowness of the domain focus reduces the number of components that need to be created to populate a reuse library and increases the probability that any given component in that library will be reused.

2.2. An example

A good example of the domain reorientation in development systems (and therefore, a good example of a successful reuse system) is the Microsoft Visual Basic™ development system [3]. It provides a visual design metaphor that allows the user to construct most of an application's user interface, database management system interface, windows/operating system services, communications services and device services via a process that is mostly cutting and pasting of domain oriented icons (e.g., text boxes, lists, dialog boxes, images, command buttons, timers, and many others). With each icon comes a set of reusable management software that provides the run-time behavior for the particular kind of object

Not only are the nature and structure of the building blocks determined by the problem domain but the organization of the software and the support tools (e.g., navigation aids and editors) are determined by the structure of the domain, not by a software engineering view of the software. There is no mechanism for presenting CASE-like design views of a Visual Basic application. However, there are quite sophisticated mechanisms for showing the design in problem oriented, visual design metaphor terms. In fact, the design screens resemble the run-time time screens as closely as feasible[1]. And objects that are graphically close or related in the design view are organizationally and navigationally close in the framework of the software. For example, in a panel design view, the code that manages a particular button click event can be accessed by double clicking on the button itself.

[1] So much so, that occasionally one forgets that he is looking at a design and tries to click a button for effect.

The Visual Basic operating metaphor is an event-driven model. In order to handle the events, the user writes small Basic functions that handle each of the events that he cares about. These functions perform much of their work by manipulating the properties of the icons (e.g., causing a panel to appear on the display by setting the "visibility" property of the panel to true). Such property manipulations are themselves events that evoke run-time behaviors.

2.3. Why now?

Many of the concepts and ideas in Visual Basic are not new. They have appeared in other similar systems before. What is new is that they are now part of the desktop computing mainstream and have been evolved to a level of operational sophistication and integration with other tools that previous attempts have not. Why has this happened now? What are the factors that have allowed or fostered this transition?

Certainly, the market has been a precipitating factor due in some measure to the desktop revolution, which has made the development of such systems economically feasible. For years, segments of the profession programming community have been searching for ways to produce simple applications, prototypes and one-offs more quickly and cheaply. And for those whose applications fit the profile of applications with heavy user interface, common stereotypical architectures (e.g., database-centric) and small amounts of additional logic, Visual Basic and systems like it provide a solution. While eventually probably 80% of the applications written will fall into this expanding profile, there will always be a small percent of custom applications that just do not lend themselves to such tools. So, while we can expect that the domain revolution will be sweeping, it will never be complete. Nevertheless, we can expect that it will indeed represent the mainstream of application development in the course of the next decade.

But the advent of the domain reorientation needed more than just market pressures to arise now. It also needed sufficiently mature supporting technologies. It would not have been possible in a few short years to both work out from scratch an understanding of the needed technologies and also develop a product that incorporated that understanding. The user interface metaphor, for example, had to be sufficiently understood and evolved. It had to evolve through the test tube of the last twenty years in order to select out the important, consistent and useful ideas of the windows metaphor. Further, developers had to understand how to engineer the parts and pieces of such applications, and their run-time support. For example, it would be impossible to invent a database interface before the architectures of database management systems were understood, experimented with and used over a period of years.

But Visual Basic and similar products of today are only the start of the domain reorientation. Today there are only a few domains that are technologically mature enough and have evolved visual metaphor conventions for their domains. Today's technologies that fit this pattern are user interface designers, DBMS interfaces, electronic forms, OS services and device services. Tomorrow there will be many more including transaction-like electronic forms, mail, groupware, telephony, digital communications, multimedia, etc. The market must grow sufficiently large to make development of the underlying technology profitable and the technology areas must mature to the point where feasible architectures are well understood.

The main open problems in domain reorientation are engineering problems -- engineering of specific domains so that they can be included in subsequent, expanded domain oriented development systems.

3. Representation abstraction

Domain reorientation provides improved reuse, more directly understandable program representations and more efficient construction systems. At first glance, it would seem that the domain reorientation has pretty much solved many of our development and maintenance problems. And indeed, we can expect that domain orientation will provide profitable improvement for many years to come. However, it does have certain weaknesses that define the next research problems to be worked on.

The shortcomings of domain oriented programming are:

- **Coverage:** It covers only a part of needed programming activities.
- **Scale:** Handling large scale programs (i.e., hundreds or thousands of KLOCs) becomes difficult.
- **Run-Time properties:** The run-time properties (e.g., performance) of the components are fixed because the components are concrete.

Eventually, we can expect that domain oriented programming will be the method of choice for perhaps as much as 80% of all programming. However, it is likely that there will always be some portion of programming that is pushing the state-of-the-art in one or more areas

and therefore, does not lend itself to domain oriented programming. Examples within today's world are:

1) high performance graphics, which is constantly reinventing itself in conjunction with evolving hardware devices, and

2) the merging of the telephone, the personal computer, the fax machine, the pager and the television, which is an area that is still defining itself.

In both of these cases, the hardware and therefore, the software architectures are likely to be completely revised several times in the next few years as the market place defines what is salable and the engineers define what is buildable.

Scale is a knotty problem to domain oriented systems but one that can be incrementally resolved through the improvements in hardware that are already aggressively in progress. We can expect constant improvement in this area as the price/performance of computer memories, hard disks and rewritable CD's improves. Of course, there will also be some Parkinsonian effect in that the need will always grow to exceed the capacity. But the problems that software can solve in this area are proportionally much less significant than those that hardware can solve.

The first two problem areas (coverage and scale) are largely a matter of engineering invention driven by market needs. There are no sweeping research breakthroughs required to solve these problems. Their solution is more a matter of evolution than revolution. However, the last problem -- run-time properties -- needs revolutionary insights.

The fixed, concrete nature of the components typically used by domain oriented development systems reduces the reuse benefits through compromising the operational characteristics of the components. Sometimes, it prevents the use of domain oriented systems altogether. Consequently, there is an open research problem -- find representations that allow increased levels of abstraction over those found in today's programming languages. In other words, find representations for reusable components that allow many of the operational characteristics to be deferred until reuse time.

3.1. Limitations of representation

Why should concreteness have such a deleterious effect on reuse? Because concrete representational forms require implementation oriented details that are needed by the compiler but could be deferred until later in the design process. Worse, premature introduction of implementation details (which often represent arbitrary design choices made in the absence of definitive requirements) precludes many opportunities for reuse. How often have you heard "I should be able to reuse this component but it is just too slow (or large, or a variety of other factors) for my application." More often than not, such reasons are perfectly valid and a potential reuse is lost because concrete, implementation details have been introduced too early -- before the opportunity for reuse, not after.

This premature introduction of implementation details is a direct result of the representations available for expressing reusable components -- conventional programming languages. Today's programming languages are largely concrete and therefore, make abstraction difficult and limit its form and degree. Let us consider the modes of abstraction that are available to the builder of a reuse library -- classes (e.g., in Smalltalk or C++), macros (e.g., in C) , generics (e.g., in Ada) and templates (as described in [5] or as defined in the C++ language).

Classes are conventionally thought of as abstractions and that term is even applied as a synonym for object oriented classes. But classes are implementation-oriented components. Their detailed algorithms are chosen and although hidden, these show through to the application in terms of their performance, size, error handling design, memory management schemes, etc. [2]. That is, the information may be hidden but the implementation properties show through and it is these properties that can have great (generally, negative) effect on the reusability of the components. So, while object orientedness is highly beneficial to reuse, it imposes built-in limits on the degree to which objects may be reused and thereby, on the expected payoff through reuse. OK, what about macros?

Macros certainly have the potential to be powerful tools but they are limited by the design considerations of the languages in which they are embedded. That is, programming languages are designed according to principles that are somewhat antithetic to reuse. Take C for example. The language was designed to allow the maximum flexibility to the programmer not the maximum abstract-ability of the code. So macros in languages like C are quite limited and have little to offer as a representation for reuse.

As an example of the key limitations of macros in languages like C, consider the requirements of generative reuse architectures. Generative reuse architectures often conditionally generate alternative code streams based on the inferred type of, or on a declared property of a data item, and typically, they apply such conditional generation recursively. C macros allow neither capability let alone

allow it to be applied recursively. Consequently, while quite useful, C-like macros are far too limited as a reuse representation candidate.

Generics (as in Ada) and templates (as in C++) add a powerful level of abstraction over simple macros because they allow components to be parmeterized on data types but they too fall short of our abstraction needs. They do not allow highly abstract components to vary based on properties that fall outside of the type system. For example, consider two coupled design decisions -- the choice of the implementation data structure for a very long strings (i.e., choosing between an array versus linked list implementation) and the choice of the substring search algorithm (e.g., choosing between a linear search versus Knuth-Morris-Pratt algorithm)[2]. The performance consequences can be onerous if the choice of implementation data structure and the choice of search algorithm are made independently, without considering the performance interactions[3]. Therefore, when I am choosing the implementation data structure for the string, I would like to be able to have my generation algorithm test the implementation property of the string search algorithm (i.e., is it linear search or KMP) and under some conditions to revise the choice of search algorithm. Types are not a good way to encode that information but extra-linguistic properties associated with the program are and this is the way Draco [4] deals with this kind of abstraction problem.

3.2. Expectations

In conclusion, if reuse is to escape the bounds of concrete representations it must include the ability to develop a rich representation of the target program that goes beyond today's programming languages (e.g., allows arbitrary extra-linguistic properties on any structure) and allows general manipulation and reorganization of the program at the level of Draco.

If implementation details are truly deferred in reusable components, then the control skeletons of many low level algorithms within those components (e.g., the string search algorithm from the earlier example) may not exist in any concrete form at the time that the component is entered into a reuse library. The control structures and their details may only be generated in concrete form when the component is reused within a specific application context. In short, abstract representations that can truly defer implementation details must by necessity be coupled to powerful generation systems that can derive much of the detail concrete structure with only a minimal involvement from the user. And it is only by such abstraction and generation that reuse can surpass the limitations that currently restrict its payoff.

4. Declining role of conventional programming languages

4.1. The effect of domain reorientation

As noted earlier, the character of development environments and processes of the last few decades has been driven by the conceptual model of conventional programming languages. For example, many popular design representation schemes (e.g., Booch diagrams) are abstractions of programming language-like structures. Similarly, the separation of the design and coding processes is a consequence of the fact that programming languages require so much concrete detail for large programs that the process of creating and organizing a program needs to be broken down into separate steps. Ideally, the first step (i.e., design) creates broad, abstract structures and the second step (i.e., coding) fills in the details.

In the next decades, the role of conventional programming languages will diminish driven by the growth of problem domain oriented design systems and the forces of abstraction. Conventional programming languages are not going to disappear. It is just that their dominant role will be passed to the domain oriented design systems. Programming languages will no longer be the main determiners of the nature of development environments and the associated processes. They will be used in a subordinate role to fill in the details of the application frameworks that are provided by the problem domain design systems.

From a process point of view, the application frameworks supplied by the problem oriented design systems represent a pre-constructed, abstracted application design. In a sense, they are a pre-cooked, generalized design that is the analog of a conventional design. And the run-time support that comes with them means that much of the coding of that broad, abstract design structure that we call high level design has already been completed. The

---

[2] This example is due to Jim Neighbors.

[3] The advantage of the KMP search algorithm arises out of the ability to avoid comparisons for many substrings (i.e., the ability to jump over some substrings) within the long search string. A linked list implementation eliminates most of this advantage and makes sequencing through the strings an expensive operation.

only coding left is the details of the application's computation.

Visual Basic and similar systems provide an example of this phenomena. The user written functions that manage events are often less than 10 or 20 lines of code and no additional code need be written to integrate them into the rest of the application. Integration is automatic as a consequence of the event-driven framework and the supporting run-time event manager.

4.2. The effect of abstraction

I also expect that nature of programming will change in that some of the information (e.g., the commitment to implement a string as an array) that was incorporated directly into algorithms written in conventional programming languages will be supplied in extra-linguistic ways to tools that generate those implementation details.

For example, many implementation details like the choice of arrays versus linked lists will not be made by the programmers directly but will be generated after the fact from the abstractions supplied by the programmers. Thus, detailed algorithmic steps that are predisposed to one or another implementation form (e.g., incrementing an array index or getting the next list item) will be abstracted away. When the programmer writes the code for a string search it will likely be a call to a generic search routine. Exactly which algorithm the generator finally chooses for that search will depend on the specific type of the data structure (much like conventional generic functions) but in addition, it may also depend on properties that exist outside of the programming language type system (e.g., requirements that imply the need for high search performance and characteristics that are consistent with a KMP style of search).

This shift toward generation based on the use of information from outside of the programming language defers the choice of implementation details and thereby, allows optimizations in implementations to be generated later in the development process -- at the time a component is incorporated into an application rather than at the time the component is incorporated into the reuse library. Consequently, we can expect:

- highly general components that are more broadly reusable (in contrast to the concrete components of many of today's libraries),
- accelerated development with fewer defects (which is a consequence of better reuse), and
- improved operational properties of the generated applications (e.g., hand tuned performance).

5. Domain driven evolution of CASE

Today's CASE tools suffer from a lack of conceptual integration with the development environments that are used to create the target programs. For example, while CASE representations are often connected to their target program's code by some level of automation, the connection is clumsy at best. Sometimes the code representation is included in the CASE design model but it often is little more than a physical embedding of code with little if any simplification or benefit gained from it. Operationally, in such cases, it is clear that two distinct representations are present (i.e., the CASE design model and the code) and the user must explicitly deal with both of them. For example, some CASE tools allow the programmer to design elements of his target program's user interface but then require that he explicitly generate the user interface code and manually integrate that code with the remainder of his application. It is not clear that the developmental effort is really simplified or decreased by such connections.

The straightforward approach to this problem would be an attempt to do a better engineering job in the integration of CASE tools with development environments. But such an approach would still be based on programming language and software engineering models of development. It would not address the integration of the domain oriented models. There is a large mental leap from software engineering representations to problem domain representations because these two representations are organized along differing conceptual dimensions. When the programmer scales up from software engineering's fine grained, general entities and relationships to the large grained, specialized entities and relationships of the problem domain, there is a paradigm shift. It is not just a matter of moving from smaller to larger. There is a fundamental shift in the way the program knowledge is organized and structured, and in the way in which the programmer deals with that knowledge. Consequently, the major challenge in CASE tools and development environment integration, is finding representations and operating regimes that emphasize the domain model and allow the programmer to operate completely from the problem point of view.

So, the question is what will the integration of CASE tools and the development environment look like in a ten years? I believe that the visual domain oriented programming systems of today provide a clue. The integration will have the following characteristics:

- There will be a single, integrated representation of the program and it will be the only representation that is visible to the programmer[4].
- It will be organized around a domain oriented framework.
- It will support visually based, direct manipulation of the domain oriented program representation.

In other words, I am suggesting that CASE tools will be reinvented. They will merge with or evolve into problem oriented design editors that today implement a cut-and-paste style of programming. The difference between a program and its design will be largely invisible to the programmer. The programmer will be able to access what ever programming details are necessary to implement, test or change the program but other details will be invisible. Further, the problem domain oriented visual metaphor will be the program's design and the programmer will understand the program abstractly in terms of the visual metaphor.

This prediction implies an evolution toward specialization into multiple visually oriented domains. Like the clip art we can expect to see "clip domains" that can be loaded into visual design editors and merged in the context of a cut-and-paste style of programming. This will require all of the other changes we have discussed in order to allow domains to be painlessly integrated and to allow the implementation details to be correctly generated.

Certainly, there have been previous efforts at integration of program representations (e.g., requirements, design and code), often around the definition of a general repository, and these efforts have not been highly successful. Why should I expect such an integration to succeed now? There are two main reasons: 1) there is a different, more problem relevant model driving the integration and 2) the technologies in the domain areas are sufficiently mature to force the integration. In the past, the repository was the general model around which the integration was defined and the varied contents of the repository were often sketchily defined and little more than placeholders (i.e., technological details to be added here.) Today, the details of a number of key domains have been worked out, in the sense that the details of design tools, APIs (Application Program Interfaces), run-time support software and so forth exist. This kind of concreteness provides a place to start with the model definition.

[4]We mean single representation in the context of a given set of tools , not a single standard for all of the world. There will almost certainly be a small number of such standards during this evolutionary change.

6. Conclusions

Perhaps the single most sweeping change will be wrought by the domain reorientation that changes the developer's conceptual model, changes the entities and relationships that the developer is dealing with, changes the enabling construction technology, and moves the developer farther away from the coding details. In short, it moves the developer farther away from conventional programming development and closer to the problem.

7. References

1. Ted J. Biggerstaff, Microelectronics and Computer Technology Corporation, "An Assessment and Analysis of Software Reuse," in ***Advances in Computers,*** Vol. 34, Academic Press, 1992.
2. Lucy Berlin, "When Objects Collide," ***OOPSLA***, 1990.
3. ***Microsoft Visual Basic™ Programmer's Guide***, Version 3.0, Microsoft Corporation (1993).
4. James M. Neighbors, "Draco: A Method for Engineering Reusable Software Systems," in ***Software Reusability***, Addison-Wesley/ACM Press, 1989.
5. Dennis Volpano and Richard B. Kieburtz, "The Templates Approach to Software Reuse," in ***Software Reusability***, Addison-Wesley/ACM Press, 1989.