The Limits of Concrete Component Reuse

Ted J. Biggerstaff

Microsoft Research

One Microsoft Way

Redmond, WA 98052-6399

Tel: (206) 936-5867

Email: tedb@microsoft.com

Abstract

This paper takes the position that the formal representation options available today for expressing reusable components -- notably, programming languages -- are excessively concrete (even with all of their advertised abstraction facilities such as classes and generics). And this concreteness imposes a built-in barrier to widespread and high payoff reuse of those components. The proposed solution to this problem is representations that allow improved deferral of implementation details (i.e., additional levels of abstraction) and allow for the automatic generation of those details at component reuse time.

Keywords: Concrete components, repositories, limitations, payoff.

Workshop Goals: To explore the notions of new representational abstractions for reuse components and the concomitant requirement for generative architectures to support those abstractions.

Working Groups: Domain analysis/engineering representations.

1 Background

I have done research on reuse and related topics (e.g., design recovery) since 1983 and this has resulted in a number of books, papers and systems. Most recently, I have been working in the area of representations that can enhance the reusability of components and yet allow most of the reuse to happen automatically. This work has led me to consider the limitations of the current concrete (i.e., programming language) representations for reusable components.

2 Position

2.1 Hypothesis

Reuse in its various forms has had a number of notable successes[1] and a number of apparent failures. In a previous article [2], I analyzed the key factors that lead to reuse success or failure. These included factors such as the scale of the component, the narrowness of the application domain, the minimization of the component interconnection architecture (e.g., the data flow structures) and the standardization of the data items (i.e., the data structures that flow between components) shared among the components in a library. But there is another factor that was not discussed in that article and one that introduces serious limits on reuse. That factor is the level of concreteness (or lack of abstraction[2]) of the reusable component.

2.2 What are Concrete Components?

Concrete components are those that express the operational meaning of the component in enough detail that it can be translated into an implementation oriented form such as a compiled function or object with little or no additional information. To the degree that a component needs additional information and to the degree that that additional information will cause extension and/or restructuring of the resulting implementation form, we will consider that component to be abstracted to some degree. For example, macros, templates and generics represent kinds of components that require additional information and exhibit extension and/or restructuring in the process of achieving their final implementation form.

Another test for concreteness is the amount of generation used by the reuse system in the course of developing the resulting implementations. If their is very little generation required and components are mostly just plugged together, then the components are truly concrete.

Generally, concrete components are expresses in conventional programming languages which by their very definition foster concreteness. In fact, I believe that the representations available today for representing reusable components are the central reason for the limitations inherent in today's reusable libraries.

1 Problem specific application generators such as GENESIS [1], more general application tool kits such as Visual BasicTM, fourth generation languages such as dBase, and many others.

2 I am using abstraction in its most general form and thus, while our use includes object-oriented programming as a special case, we intend abstraction to have its general, non-jargon meaning.

2.3 Limitations of Representation

Why should concreteness have a deleterious effect on reuse? Because concrete representational forms require implementation oriented details that are needed by the compiler but could easily 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 -- 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 general templates (as described in [5]).

Classes are conventionally thought of as abstractions and that term is even applied to describe them. But classes are already 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. [3] That is, the information may be hidden but the implementation properties show through and it is these concrete 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, let's look further. 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 antithetical 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 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 are an improvement on C-like macros. But generics can only vary based on data types and, in Ada for example, only on the built-in types. This limitation to built-in types is probably in Ada because the emphasis in the design of the Ada language was validation based on type consistency which would mean that the compiler would have to have the type inference rules built-in so that it could do type inference efficiently. But the downside of this limitation is that one cannot abstract generics to the degree desired. Even abstraction based on user defined types would help but it too would fall short of the needs of abstraction for

reuse. Consequently, while generics are a step in the right direction, they do not go far enough. OK, what about general templates that vary based on types including user defined types.

Templates add a powerful level of abstraction but they too fall short of what is needed simply because highly abstract components must 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). 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 general properties associated with the program are and this is the way DRACO [4] dealt with this kind of abstraction problem.

2.4 Representational Abstraction

In conclusion, if reuse is to escape the bounds of concrete representations it must move beyond static, 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.

3 Comparison

This view of reuse falls somewhere between the fully generative view where all of the componentry is completely built into the generation system and the fully compositional view where all of the componentry is explicitly and completely defined by the end users[4] of the system. Visual Basic™ is a good example of a fully generative reuse system and the Foundation Classes for Microsoft's C compiler, which defines a set of reusable components that simplify Windows™ programming, is a good example of the fully compositional

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.

4 Or at least personnel that are closer to the end user than to the developer of the reuse system. The author recognizes that the population of the reuse library is frequently not done by end users (i.e., application developers) but rather by domain analysts.

view. The view of representation that is most similar to the view expressed in this paper is that of Neighbors [4]. Within Neighbor's DRACO system, there is a set of clearly identifiable componentry (expressed in domain oriented languages) that the programmer can change and manipulate, but there is also a generative mechanism (i.e., a transformation engine) that is driven by some of the components (i.e., those that are expressed as transformations). This architecture allows the DRACO user to develop components that are far more abstract than possible with current programming languages.

While the view of this paper targets roughly the level of component abstraction in DRACO, it differs from the DRACO view in a couple of ways. It accepts a larger user involvement in the decision making that controls the generation of the concrete implementations and therefore, it anticipates a generative system that is somewhat less general and less automated than the DRACO model. The details of such a model and the generative architecture that it requires await the answers to the representation research, which will determine whether or not this ideal is achievable.

References

[1] Batory, D.S., "Concepts for a Database Compiler", *ACM PODS*, 1988.

[2] Biggerstaff, T.J., Microelectronics and Computer Technology Corporation, "An Assessment and Analysis of Software Reuse", in *Advances in Computers*, Vol. 34, Academic Press, 1992.

[3] Berlin, L., "When Objects Collide", *OOPSLA*, 1990.

[4] Neighbors, J., "DRACO: A Method for Engineering Reusable Software Systems", in *Software Reusability*, ACM Press, 1989.

[5] Volpano, D. and Kieburtz, R.B., "The Templates Approach to Software Reuse", in *Software Reusability*, ACM Press, 1989.

4 Biography

Ted J. Biggerstaff is Research Program Manager of the Intentional Programming Project at Microsoft Research Laboratories. He is responsible for exploring ways of enhancing reuse capabilities of application development. Before coming to Microsoft, he was Director of Design Information at MCC where he led the research and development of the DESIRE design recovery system. Before that, he worked for ITT's programming laboratory (on reuse) and Boeing Computer Service's Advanced Technology and Application Division (on CASE tools). He received a Ph.D. in Computer Science from the University of Washington in 1976.