[Bla91] L. Blaine and A. Goldberg, ‘DTRE - A Semi-Automatic Transformation System’, in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.

[Hei94] J.S. Heidemann and G.J. Popek, ‘File-System Development with Stackable Layers’, *ACM TOCS*, 12(1), 58-89.

[Joh88] R.E. Johnson and B. Foote, ‘Designing Reusable Classes’, *Journal of Object-Oriented Programming*, June/July 1988.

[Kic93] G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1993.

[Per89] D.E. Perry, “The Inscape Environment”, *Proc. ICSE 1989*, 2-12.

[Tra93] W. Tracz and L. Coglianese, ‘An Adaptable Software Architecture For Integrated Avionics’, *Proc. NAECON*, May 1993.

5 Biography of Instructor

Don Batory is an Associate Professor in Computer Sciences at the University of Texas at Austin. He was a member of the ACM Software Systems Award Committee from 1989-1994. He was an Associate Editor of *ACM Transactions on Database Systems* from 1986 to 1992, and an Associate Editor of *IEEE Database Engineering* from 1981 to 1984. He was the Program Committee Chairman for the *3rd International Conference on Software Reuse* (November, 1994). His research interests are in software system generators, domain modeling, transformation systems, and database management systems.

perspective, components are generally not individual classes, but are subsystems - i.e., suites of interrelated classes. Thus, a component's export interface typically consists of multiple classes and a component itself encapsulates the implementation of these classes. A *realm* is a library of plug-compatible components.

GenVoca realms and object-oriented frameworks are related [Joh88, Bat92], although the exact relationship is not yet fully understood. An essential aspect of framework design to express collaborations among abstract classes as "template" operations; it is not obvious where this counterpart lies in GenVoca. Our work has focussed on developing a clean abstract model of component composition; we do not yet know how the composability of frameworks relates to our model.

Components that are used by generators rely on a particular form of encapsulation. GenVoca components are essentially forward refinement program transformations [Bla91] that encapsulate consistent data and operation refinements, as well as reflective computations on metadata [Kic93]. The concept of data and operation refinements is familiar to us all: it is the function and class encapsulation that is provided by object-oriented programming languages. In order to generate efficient code, however, it is absolutely essential for some domains that metadata (i.e., information about the algorithms or classes that are being encapsulated) and computations on metadata (i.e., reflective inferencing to decide when to use a particular algorithm or apply a certain domain-specific optimization) also be encapsulated within a component. Thus, design concerns for components may be different than those of recognized OO methodologies.

There are other unusual characteristics of GenVoca components and domain modeling that we have encountered. If a system can be defined by a composition of components, how does one know if a particular composition meets the requirements of a target system? This is a problem of design rule checking or consistency checking [Per89, Bat95b]. Additionally, GenVoca components have the unusual property that their interface can enlarge upon instantiation. That is, when a component is instantiated, new functions and classes can automatically appear to be part of its interface [Hei94, Bat95a]. We know of no counterpart to this phenomena in object-oriented technology.

3 Next Steps

Domain modeling and software system generation are exciting and challenging areas of software engineering research. The techniques and concepts that we have found in prototype generators is an unusual meld of ideas from object-oriented programming, reflective programming, parameterized programming, program transformation systems, and software architectures. Exploring these relationships and developing general tools (see [Tra93]) for creating domain models and generators are important goals for future research.

4 References

[Bat92] D. Batory and S. O'Malley, 'The Design and Implementation of Hierarchical Software Systems with Reusable Components', *ACM TOSEM*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, 'Scalable Software Libraries', *Proc. ACM SIGSOFT 1993*.

[Bat95a] D. Batory, 'Software System Generators, Architectures, and Reuse', Tutorial Notes.

[Bat95b] D. Batory and B. J. Geraci. 'Validating Component Compositions in Software System Generators', Technical Report TR-95-03, Department of Computer Sciences, University of Texas at Austin, February 1995.

Issues in Domain Modeling and Software System Generation[1]

Don Batory
The University of Texas at Austin
batory@cs.utexas.edu

1 Introduction

Years from now, much of the software that we are developing today and the software design and implementation problems that we are now facing will be so well understood that it will be possible to automate the development of such software for large families of applications. This will be accomplished through the use of software system generators. Such generators will automatically transform compact, high-level specifications of target systems into actual source code, and will rely on libraries of plug-compatible and reusable components for code synthesis.

Over the last ten years, our research has focussed on domain modeling methodologies and implementation techniques for creating software system generators. To us, a *domain model* is a component-based, parametric definition of a family of related systems. Every family member is defined by a unique composition of components. The similarities and differences among different family members are exposed by comparing the component compositions that define them. An implementation of such a model is a *software system generator.*

The modeling methods and implementation techniques that we have developed are expressed in the *GenVoca* model [Bat92]. GenVoca concepts have been distilled from the experiences of building software system generators for the disparate domains of database management systems, communication protocols, avionics, file systems, and data structures [Bat92-93, Tra93, Hei94]. Common characteristics of generators are that the performance of generated systems often match or exceed that of hand-written code (because generators can perform optimizations automatically that are difficult, if not impractical, to do by hand) and that substantial productivity gains (e.g., factors of 3-4) are delivered through the reuse of components.

Among the basic goals of our work is to provide software engineers with proven conceptual abstractions and implementation techniques that are needed for programming-in-the-large, i.e., specifying large-scale systems as compositions of components, and analyzing and reasoning about systems in terms of their components. This entails raising the level of programming from writing individual lines of code, functions, and classes, to that of composing subsystems. This, in turn, appears to require extensions to or adaptations of conventional object-oriented concepts and design techniques. Identifying these extensions (or differences) is an important aspect of our work.

2 Relationships with Object-Orientation

The GenVoca approach to software system generation relies on standardized and layered decompositions of software systems. The building blocks of software systems are components, which are layers. Components both export and import “standardized” interfaces so that they act like software “legos”. That is, target software systems are hierarchical compositions of plug-compatible components. From an object-oriented

1. This research was supported in part by the Applied Research Laboratories of the University of Texas at Austin, Schlumberger, and Microsoft.