D. Batory

Product-Line Architectures

Invited Presentation, *Smalltalk und Java in Industrie und Ausbildung* (*Smalltalk and Java in Industry and Practical Training*), Erfurt, Germany, October 1998.

Abstract

Today's software design methodologies are aimed at one-of-a-kind applications, designs are expressed in terms of objects and classes, and software must be coded manually. We argue that future software development will be very different and will center around product-line architectures (i.e., designs for families of related applications), refinements (a generalization of today's components), and software plug-and-play (a codeless form of programming).

1 Introduction

In the 1500's, it was common and obvious knowledge that the Earth was the center of the Universe; all heavenly bodies — moon, sun, planets, stars — acknowledged the Earth's dominance by revolving around the Earth. For the most part, geocentricity provided an adequate model of the Universe. Predictions of lunar eclipses were amazingly accurate. (Geocentricity was correct). So too were predictions of the positions of "fixed" stars. (They didn't move). But the motion of planets was problematic because they did not traverse the sky in simple ways; planets would periodically move backwards against the background of stars before continuing their forward motion. (Today we call this retrograde motion). Scientists of that era proposed inscrutable models that utilized rotating nested spheres to explain retrogrades; but ultimately these models failed to predict planetary motions accurately.

In 1543, Copernicus proposed a radically different explanation of the Universe by recognizing the heliocentric nature of our solar system. Not only did heliocentricity explain retrograde motions in a simple and elegant manner, it laid the foundation for today's understanding of planetary systems. Copernicus's result is an extreme, but clear, illustration of how science progresses. That is, by negating commonly held "truths" yields models of the Universe that are not only consistent with known facts, but are more powerful and lead to deeper understandings and results that simply could not be obtained otherwise. The scientific results that I and many others have made in our careers are more common examples of this paradigm, i.e., results that led to incremental advances.

Today we live in a universe of software. Software is elegantly explained in terms of objects that are instances of classes, classes are related to other classes via inheritance, and webs of interconnected objects accurately model run-time executions of applications. Object-orientation has revolutionized our understanding of software. We have abandoned a function-centric view of software where functional decomposition guided our understanding of appropriate encapsulations to a data-centric view where object/class encapsulations reign supreme. The OO view of software is indeed very powerful and will remain current for some time. But years from now, will we have an alternative view of software? The answer, of course, is yes. But what will it be? What will replace the abstractions of objects and classes? How will we produce and specify software? Our clairvoyance is guided by negating some obvious contemporary "truths" and seeing if a consistent explanation of our software universe remains.

Consider the following three "truths":

- contemporary OO design methodologies are aimed at producing one-of-a-kind applications,
- application designs are expressed in terms of objects and classes, and
- we manually code our implementations given such designs.

Each of these points belabors the obvious. At the same time, it is not difficult to envision change. Future design methodologies will not focus on unique applications but rather on families of related applications, called *product-line architectures (PLAs)*. Designs of PLAs will not be expressed purely in terms of objects and classes, but rather in terms of components. Actually, expressing software designs in terms of components is already a contemporary phenomenon; so in this paper we anticipate the next step beyond today's components called *refinements*. And finally, application development will exploit *codeless programming*. Both industry and academia are moving toward *software plug-and-play* — i.e., the ability to assemble applications of a product-line quickly and cheaply merely through component composition; no source code has to be written.

These ideas are further motivated and clarified in the following sections.

1.1 Product-Line Architectures

A *product-line architecture (PLA)* is a blue-print for creating families of related applications. PLAs acknowledge the fact that companies don't build individual products, but instead create families of closely related products. The importance of PLAs is evident: software complexity is growing at an alarming rate and the costs of software development and maintenance must be restrained. PLAs enable companies to amortize the effort of software design and development over multiple products, thereby substantially reducing costs.

Recognizing the need for PLAs is ancient history: McIlroy motivated PLAs in 1969 [McI69] and Parnas described the benefits of software families in the mid-1970s [Par79]. What has changed from the time of these pioneering predictions is that proven methodologies for building and understanding PLAs are available. In fact, the first steps in evolving contemporary one-of-a-kind OO design methodologies toward PLAs has occurred. Jacobson, Griss, and Jonsson [Jac97], for example, advocate *variation points*, i.e., one or more locations at which variations will occur within a class, type, or use case. Different application instances will utilize different variations, which is clearly the beginning of a product-line architecture.

1.2 Generalizing Components to Refinements

Today's newest object-oriented methodologies are not based purely on objects and classes, but on components. A *component* is an encapsulated suite of interrelated classes. Its purpose is to scale the unit of design and software construction from an individual class to that of a package or OO framework. The most recent design methodologies from Catalysis [D'S98] and Rational [Rat98], for example, are explicitly named "Component-Based Software Designs", where components could be OO packages, COM or CORBA components, Java Beans, and so on.

To give readers a perspective of where work on software componentry is headed, let me share my experiences of working for fifteen years on component-based product-line architecture methodologies. I have encountered many domains where components simply cannot be implemented as OO packages or as COM or CORBA components. The reason is one of performance: applications that were constructed with these components were so horrendously inefficient that no sane person would ever use them. Does this mean that components can't exist for these domains? Certainly not — if anything, building applications from components is a goal that we want to achieve for *all* domains. What it actually means is that the components of these domains must be implemented differently. Implementations must break component encapsulations for domain-specific optimizations. Instead of equating components with "concrete" source code that would be statically or dynamically linked into an application, a more appropriate implementation might be as a *metaprogram* — i.e., a program that generates the source the application is to execute by composing prewritten code fragments.[1] Or if metaprograms are not sophisticated enough to produce efficient source, a component might be implemented as a set of rules in a program transformation system [Par83, Bax92]. (A *program transformation system* is a technology by which program specifications are transformed into effi-

cient source by applying semantically-correct program rewrite rules; code motion and complex optimizations are examples).

Given this observation, I realized that today's notions of components are simply too implementation-oriented or implementation-specific. We need to separate a component's *abstraction* from its possible *implementations*, where OO packages, COM, metaprograms, program transformation systems are merely different component implementation technologies, each with their own competing strengths and weaknesses. The component abstraction that we seek that unifies this wide spectrum of technologies is that of a *(data) refinement*.

What is a refinement? I'll give an informal example now and a more precise definition later. Have you ever added a new feature to an existing application? What you discover is that changes aren't localized: multiple classes of an application must be simultaneously updated (e.g., adding new data members, methods, replacing existing methods with new methods, etc.). All of these modifications must be made consistently *and* simultaneously if the resulting application is to work correctly. It is this collection of modifications — called a refinement — that defines a "component" or "building-block" for that feature. By analogy, removing this component/feature from an application requires all of its modifications to be simultaneously and consistently removed. (How such refinements are encapsulated and realized is considered in Section 4.2).

This line of reasoning led me to conclude that refinements are central to a general theory of product-line architectures. Abstracting away component implementation details reveals a common set of concepts that underlie the building blocks of domains and product-line architectures. The benefit in doing so is a significant conceptual economy: there is one way in which to conceptualize the building blocks of product-line architectures, yet there are multiple ways in which blocks can be implemented. The choice of implementation is ultimately PLA/domain-specific.

1.3 Software Plug-and-Play

Programming with today's components is analogous to the old-fashioned way circuit boards were created decades ago: one collects a set of chips, plugs them into a board, and wire-wraps connections between chip pins to implement a particular functionality. In essence, this is *the* classical library paradigm: one collects a set of software modules and writes "glue" code to interweave calls to different modules to implement a particular functionality. This has always been a very successful and largely manual process for creating customized hardware and software applications.

It is obvious to most people that software construction allows for much greater degrees of automation. It should be possible to "drop" a component into an existing system and — by imbuing the component with domain-specific expertise — have it "wire-wrap" its connections and thereby automate a tedious coding process that experts would otherwise have to do manually. Hardware plug-and-play is a practical realization of this idea. Today we can customize PC configurations merely by connecting components. Although the connections are simple for us, a myriad of low-level connections are being made via standardized hardware interfaces. Hardware plug-and-play makes PC extensions and reconfigurations almost trivial and has empowered novices to do the work once reserved for high-paid experts. We need the same for building and extending software applications by plugging and unplugging components with standardized interfaces.

1. Metaprograms can be implemented in OO languages. My point is that a metaprogram is *not* application source code, but is *considerably* more abstract and fundamentally different: it is a *generator* of customized application source.

1.4 Recap

Software development will inevitably evolve toward product-line architectures, distinguishing component abstractions (refinements) from their implementations, and software plug-and-play. The challenge is to achieve these goals. In the following sections, I'll outline a particular way to do so.

2 Understanding Product-Line Architectures

There are many results that are relevant to this view of the future. Work on extensible or open systems is the first step toward creating PLAs [Obe98]. Research on "domain-specific software architectures" was to develop PLAs for a variety of military domains [Met92]. Aspect-Oriented Programming, while not specifically aimed at PLAs, certainly has much in common with the basic mechanisms that are needed [Kic97]. Subject-Oriented Programming, where different applications are constructed by composing different "subjects", clearly deals with PLAs [Har83]. Feature-Oriented Programming extends OO methodologies to product-lines [Coh98]. There are many other relevant efforts (see [Cza97]). Please note that all of these approaches (including that of Section 3) are not identical in their technical details — one shouldn't expect them to be — but the essential problems that they address are remarkably similar.

Common to most models of product-line architectures are three ideas: (1) identifying the set of features that arise in a family of applications, (2) implementing each feature in one or more ways, and (3) defining specific applications of a PLA by the set of features that it supports plus the particular implementation of each feature.

In the early 1990's, I encountered a classic example of a PLA. What attracted me to this example was the unscalability of its design. The Booch Components have undergone a long evolutionary history of improvement [Boo87-93]. The original version was in Ada containing over 400 different data structure templates/generics. For example, there were 18 varieties of dequeues (i.e., double-ended queues). How did the number 18 arise? Booch proposed a PLA where dequeue data structures had three features: concurrency, memory allocation, and ordering. The concurrency feature had three implementations (which users had to choose one): sequential (meaning there was no concurrency), guarded (programmers had to call waits and signals explicitly), and synchronized (waits and signals would be called automatically). The memory allocation feature also had three implementations (choose one): bounded (meaning that elements were stored in an array), unbounded (elements were stored on a list), and dynamic (elements were stored on a list with memory management). The ordered feature had two implementations (choose one): elements were maintained in key order or they were unordered. Because feature implementations were orthogonal, there were $18 = 3 \times 3 \times 2$ distinct variations of dequeues.

This approach had glaring problems: what happens when a new feature is added, such as persistence? The consequence is every data structure/dequeue in transient memory must be replicated to exist in persistent memory. That is, the library doubles in size! The problem is actually worse: if one examines even contemporary data structure libraries for C++ (e.g., STL) and Java, one discovers the data structures that are offered are elementary and simplistic. The data structures that are found in, for example, operating systems, compilers, database systems, etc. are *much* more complicated. What this means is that people are constantly adding new features. This led me to conclude that no conventional library could ever encompass the enormous spectrum of data structures (or more generally, applications of a product line) that will be encountered in practice. Clearly, there had to be a better way to build PLAs.

The general problem that the Booch Components exhibited was the lack of library *scalability.* Given n optional features, one has a product-line of $O(2^n)$ distinct applications. Or more generally, if each feature has m different implementations, the product-line is of size $O((1+m)^n)$. What this tells us is that libraries of PLAs shouldn't contain components that implement combinations of features. Instead, *scalable* libraries contain components that implement individual and largely orthogonal features. A scalable library is quite

small — on the order of $O(mn)$ components — but the number of applications that can be assembled from component compositions is very large — e.g. $O((1+m)^n)$ [Bat93, Big94].

To explore the possible impact of this approach, Singhal reengineered a C++ version (v1.47) of the Booch Components (see Table 1 and [Sin96]). For that part of the library to which these ideas applied, he reduced the number of "components" from 82 to 22, the number of lines of source from 11K to under 3K, and *increased* the number of distinct data structures (applications) that could be created from 169 to 208 (i.e., there were applications in Booch's product line that were unimplemented). When benchmarks were run on corresponding data structures, Singhal's data structures were more efficient. More importantly, it was very easy to add new features (i.e. components) to Singhal's library that would substantially enlarge the number of data structures that could be created; this couldn't be done with the Booch design. Clearly, this was a big win. The question then became: can these results be replicated for more complicated domains?

	Booch	**Singhal**
# of Components	82	22
Lines of Code	11,067	2,760
# of Product-Line Applications	169	208

Table 1: Comparing the Booch and Singhal Components

The answer is yes. Scalable PLA libraries for domains as varied as database systems (where applications are individual DBMSs that can *exceed* 80K LOC), protocols, compilers, avionics, hand-held radios, and audio-signal processing [Cog93, Bat93, Hei93, Hut91, Ren96]. Generally the PLAs for these domains were created in isolation of each other, which means that researchers are reinventing a common set of ideas. In the following section, we review these ideas which we have collectively called *GenVoca*; the name stems from the first known PLAs based on this approach, namely "Genesis" and "Avoca" [Bat92].

3 GenVoca

The obvious way in which to create plug-compatible components is to define standardized interfaces. GenVoca takes the idea of components that export and import standardized interfaces to its logical conclusion.

Virtual Machines. Every domain/PLA of applications has a small set of fundamental abstractions. Standardized interfaces or virtual machines can be defined for each abstraction. A *virtual machine* is a set of classes, their objects, and methods that work cooperatively to implement some functionality. Clients of a virtual machine do not know how this functionality is implemented.

Components and Realms. A *component* or *layer* is an implementation of a virtual machine. The set of all components that implement the same virtual machine forms a *realm*; effectively, a realm is a library of plug-compatible components. In Figure 1a, realms **S** and **T** each have three components, whereas realm **W** has four. All components of realm **S**, for example, are plug-compatible because they implement the same interface. The same holds for realms **T** and **W**.

```
(a)  S = { a, b, c }                 (b)  S :=   a  |  b  |  c  ;
     T = { d[S], e[S], f[S] }             T :=  d S | e S | f S ;
     W = { n[W], m[W], p, q[T,S] }        W :=  n W | m W |  p  | q T S ;
```

Figure 1: Realms, Components, and Grammars

Parameters and Refinements. Just as components can export standardized interfaces, so too can they import standardized interfaces. A component has a (realm) parameter for every realm interface that it imports. All components of realm **T**, for example, have a single parameter of realm **S**.[2] This means that every component of **T** exports the virtual machine of **T** (because it belongs to realm **T**) and imports the virtual machine

interface of **S** (because it has a parameter of realm **S**). Each **T** component encapsulates a *refinement* between the virtual machines **T** and **S**. Such refinements can be simple or they can involve domain-specific optimizations and the automated selection of algorithms.

Applications and Type Equations. A hierarchical *application* is defined by a series of progressively more abstract virtual machines [Dij68]. Its implementation is expressed by a named composition of components called a *type equation*. Consider the following two equations:

```
A1 = d[ b ];
A2 = f[ a ];
```

Application **A1** composes component **d** with **b**; **A2** composes **f** with **a**. Both applications are equations of type **T** (because the outermost components of both are of type **T**). This means that **A1** and **A2** implement the same virtual machine and are plug-compatible implementations of **T**.

Note two things: (1) composing components is equivalent to stacking layers. For this reason, we use the terms *component* and *layer* interchangeably. (2) Specifying applications as type equations achieves the codeless programming of software plug-and-play; no source code (other than the components themselves) has to be written.

Grammars, Product-Lines, and Scalability. Realms and their components define a grammar whose sentences are applications. Figure 1a enumerated realms **S**, **T**, and **W**; the corresponding grammar is shown in Figure 1b. Just as the set of all sentences defines a language, the set of all component compositions defines an application *product-line*. Adding a new component to a realm is equivalent to adding a new rule to a grammar; the family of products that can be created enlarges substantially. Because large families of products can be built using few components, GenVoca is a *scalable* model of software construction.

Symmetry. Just as recursion is fundamental to grammars, recursion in the form of symmetric components is fundamental to GenVoca. More specifically, a component is *symmetric* if it exports the same interface that it imports (i.e., a symmetric component of realm **W** has at least one parameter of type **W**). Symmetric components have the unusual property that they can be composed in arbitrary ways. In realm **W** of Figure 1, components **n** and **m** are symmetric whereas **p** and **q** are not. This means that compositions **n[m[p]]**, **m[n[p]]**, **n[n[p]]**, and **m[m[p]]** are possible, the latter two showing that a component can be composed with itself.

Why are symmetric components useful? All applications have open-ended sets of features. Symmetric components enable new features to be added to an application; they enrich the capabilities of applications (without altering the application's fundamental abstractions). For example, all relational database systems (RDBs) present users with relational abstractions. Some RDBs offer more sophisticated query languages than others. Additional querying capabilities — which symmetric components can provide — merely enrich the relational abstractions that are offered. More on this in Section 4.2.

Design Rules, Domain Models, and Generators. In principle, any component of realm **S** can instantiate the parameter of any component of realm **T**. Although the resulting equations would be *type correct,* the equation may not be *semantically* correct. That is, there are often domain-specific constraints that instantiating components must satisfy *in addition to* implementing a particular virtual machine. These additional constraints are called *design rules*. *Design rule checking (DRC)* is the process of applying design rules to validate type equations [Bat97a]. A GenVoca *domain model* or *product-line model* consists of realms of components and design rules that govern component composition. A *generator* is an implementation of such a model; it is a tool that translates a type equation into an executable application.

2. Components may have many other parameters besides realm parameters. Here we focus only on realm parameters. Also, parameterizations we will deal with in this paper are simple enough to dispense with formal parameter names.

Implementations. A GenVoca model says nothing about when components/refinements are to be composed — the options are dynamically at run-time or statically at compile-time — or how components/refinements are to be implemented — OO packages, COM components, metaprograms, program transformation systems, etc. The bindings of these implementation decisions are made *after* the model is created and is largely determined by the domain and the efficiency of constructed applications. Generally OO and COM implementations offer no possibilities of static optimizations. Metaprogramming implementations automate a wide range of common and simple static domain-specific optimizations; program transformation systems offer unlimited static optimization possibilities. Table 2 tallies the distribution of GenVoca PLAs according to their implementations. Most use a uniform component implementation and binding-time strategy. Others, like hand-held radios, optimize components that are composed statically, and otherwise perform no optimizations for other components that are composed dynamically.

	Static	**Dynamic**
OO, COM, …	databases, avionics, compilers	protocols
Metaprogramming	data structures	audio signal processing
Program Xform Systems	protocols	

hand-held radios

Table 2: Classification of PLAs on Component Binding Time and Implementation Technologies

4 Experience

GenVoca PLAs have been very successful. Performance of synthesized applications is comparable or substantially better than that of expert-coded software. Productivity increases range from a factor of 4 (where new components have to be written) to several orders of magnitude (where all components are available). Further, an 8-fold reduction in errors has been reported. See [Bat97b] for details.

There are problems and limitations with every approach, and GenVoca is no exception. Both technical and nontechnical issues abound. Experience has revealed no technical “show-stoppers”; to be sure, there are plenty of interesting technical challenges, but these are solvable. The hardest problems are nontechnical.

4.1 Nontechnical Barriers

Legacy Code. Companies have legacy software that they want to reuse in product-line applications. They are willing to accept the penalty of hacking source code (for customization, efficiency improvements) in building PLA applications. For many domains, particularly those that have little variation, this approach is reasonable. When product-lines are enormous, justifying a re-engineering of legacy source is a very hard decision to make, no matter how significant the potential benefits.

Corporate Politics. Even when re-engineering is warranted, ad hoc approaches are attractive. In my experience, it is necessary to develop prototypes to demonstrate scalable PLA benefits and capabilities, often to the point where one can run circles around competing approaches. While necessary, prototypes aren’t sufficient. Corporate politics, egos, pre-existing methods, and insecure funding can easily obscure technical goals. The basic problem is that technology adoption decisions are made largely on the basis of non-technical reasons, and the results of these decisions are often confused with technical reasons.

Thin Windows of Opportunity. Thinking in terms of layered refinements and components with standardized interfaces is both the greatest strength of GenVoca as well as its greatest weakness. Architects of PLAs may not be open to new approaches or the risks that may be entailed. Architects that are hardest to reach are often those that have recognized the need for PLAs and that have already selected their course of action. Architects that are the easiest to influence are those that have the least investment. The window of opportunity is thin: it begins after the need for PLAs has been recognized and ends when a solution approach has been adopted.

Catch 22. There are companies whose motto is "we won't use new technologies until others use them". Of course, these other companies say the same thing, leading to adoption deadlock. There is no technical reason for this standoff, only one of risk aversion.

Terminology Arms Race. Five years ago, the word "architecture" was rarely used in the software literature. Now every one seems to use it. Unfortunately, just about everyone has a different meaning (regardless of whether they've ever bothered to define the term). This makes it difficult to compare, contrast, or even categorize approaches. For example, to distinguish an approach as being "scalable product-line architectures" is largely meaningless, because everyone's approach is assuredly "scalable" and deals with "product-lines". This encourages the invention new terms (for established or possibly undeveloped concepts), rewards those with catchy slogans, and obscures the recognition of real progress for everyone.

Not Ready for Prime Time. Ideas are adopted when the time is right; there are all sorts of implicit preconditions that must be satisfied. More often than not, idea mantras have to be chanted by many people to be believed, and not just any people, but by recognized community leaders.

I recall in 1983 of a conversation with a colleague (whom today as then I greatly admire) about the possibility of creating customized database systems simply by composing plug-compatible components. His response surprised me — "That's impossible!". I realized that he was right; it was impossible for *him* to do this, but that didn't mean that *others* couldn't do it. I also learned that pointing out the fallacy of someone's reasoning rarely leads to lasting gratitude :-).

Years later I heard people say "My software is too complicated to be built in that way!". On both occasions, the projects of these individuals folded within a year. Their real statement was the preface: "My software is too complicated, period". Programmers have a seemingly infinite capacity for complicating the most simple things. Even with a minimal exposure to data refinements, it is evident that programmers are unknowingly composing refinements as they write code. Every line of their programs can be traced to a refinement in some composition of refinements. All that GenVoca is doing is making this simplicity explicit and reaping the benefits of doing so.

4.2 Technical Problems

Testing and Verification. The most challenging open problem today is testing. We can synthesize high-performance, customized applications quickly and cheaply, but questions about the validity of the generated source remain. It is still necessary to subject synthesized applications to a battery of regression tests to gain a level of confidence that it is (sufficiently) correct. The ultimate goal of PLAs is to shrink the release time for new products; it is not yet clear how PLA organizations can reduce testing (see [Edw98]).

There is hope from verification research. Formal approaches to verified software are often based on (data) refinements (e.g., [Sri96]). The Ensemble/Horus projects at Cornell, for example, are GenVoca-like PLAs for building verified distributed applications from symmetric components [Ren96, Hay98]. Individual components have been verified; so high assurance statements can be made about their compositions. Boerger's *Evolving Algebra (EV)* addresses the problem of scaling proofs for individual applications to families of related applications [Boe96]. EV is clearly based on layered refinements.

Refinements. There has been a tremendous amount of work, both theoretical (e.g., [Bro86, Sri96, Boe96]) and pragmatic (e.g., [Nei84, Bax92]), on refinements. My work on GenVoca has admittedly evolved largely in isolation from this work. The primary reason was not lack of interest, but project objectives. Most of the fodder that I used to develop GenVoca stems from my implementation projects and those of others whose goals were to explore and develop software plug-and-play PLAs. It was not in the purview or interest of these individual projects to generalize the idea of layering to other domains. However, doing so raises the connection with refinements. It is an open problem to unify theoretical results with experimental findings.

Design Wizards. Common problems that users of GenVoca PLAs encounter are not knowing what components to use or what combinations of components satisfy application requirements. Until a certain level of expertise develops, it not difficult for users to specify type equations that are semantically correct but are not appropriate (e.g., performance-wise) for the target application. An expert in the domain and PLAs would critique a proposed design by saying "Don't use this combination of components, but this combination instead for the following reasons…". This is the kind of expertise that needs to be automated.

A key to this problem is that application designs are expressed as equations. Expert knowledge of what to use and when to use components can be captured as *algebraic rewrite rules* (i.e., replace expression **x** with **y** under condition **z** because of reason **r**). By collecting such rules and using standard rule-based optimizers, a tool called a *design wizard* can be developed that will *automatically* (a) optimize equations — application designs — given workload specifications and (b) critique equations to avoid blunders. We have developed a design wizard for one domain [Bat98]. However, it is open problem to show that the approach is general enough to work for other domains.

Standardized Interfaces. Applications of a product line rarely have the same programming interface. How then can applications with variable interfaces be constructed from components with standardized interfaces? In truth, standardized interfaces do not mean cast-in-concrete interfaces. It is possible to insert a component deep into the bowels of an application, and have the exported interface of the application change. It is only recently that we have found a general solution to this problem [Sma98]. (See the next topic).

Bridging Communities. Challenging technical problems often arise due to the inability of others to understand the concept of refinements in their own terms. The reason is that refinements often require unusual juxtapositions of ideas and this, in turn, leads to interesting technological advances.

As case in point, it took us several years to understand a fundamental connection between refinements and OO — that is, how are refinements expressed as OO concepts? The answer is rather simple. A refinement of a class may involve the addition of new data members, new methods, and overriding existing methods. Such refinements are easily expressed via subclassing: a subclass (that encapsulates refinement modifications) is declared with the original class as its superclass.

GenVoca layers encapsulate suites of interrelated classes. Figure 2a shows a (terminal) shaded layer that encapsulates three classes. Figure 2b shows a darker layer that also encapsulates three classes; when it is stacked on top of the shaded layer, one class becomes a subclass of the left-most shaded class, while the others begin new inheritance hierarchies. Figure 2c shows a white layer to encapsulate four classes. When stacked upon the darker layer, each class becomes a subclass of an existing class. Lastly, Figure 2d shows the effect of adding a black layer, which encapsulates two classes. The application (which is defined by the resulting layer stack) instantiates the most refined classes (i.e., the terminal classes) of these hierarchies. These classes are circled in Figure 2d; the non-terminal classes represent *intermediate derivations of the terminal classes. Thus, when GenVoca components are composed, a forest of inheritance hierarchies is created. Adding a new component (stacking a new layer) causes the forest to get progressively broader and deeper* [Sma98]. Although straightforward, these ideas are not obvious nor are they common. It is through the use of inheritance that new operations/methods can be added to multiple application classes merely by plugging in a component.

It is possible to express the ideas of Figure 2 using mixins. (A *mixin* is a class whose superclass is specified by a parameter). We wanted a clean expression of these ideas in Java.[3] Unfortunately, neither Java or Pizza [Ode97] (a dialect of Java that supports parameterized polymorphism) supports parameterized inheritance. What we really needed was an extensible Java language in which it was possible to add other features to express refinements.

3. We chose Java because of the language's simplicity and also to more clearly show the concepts that programming languages are lacking in order to express refinements.

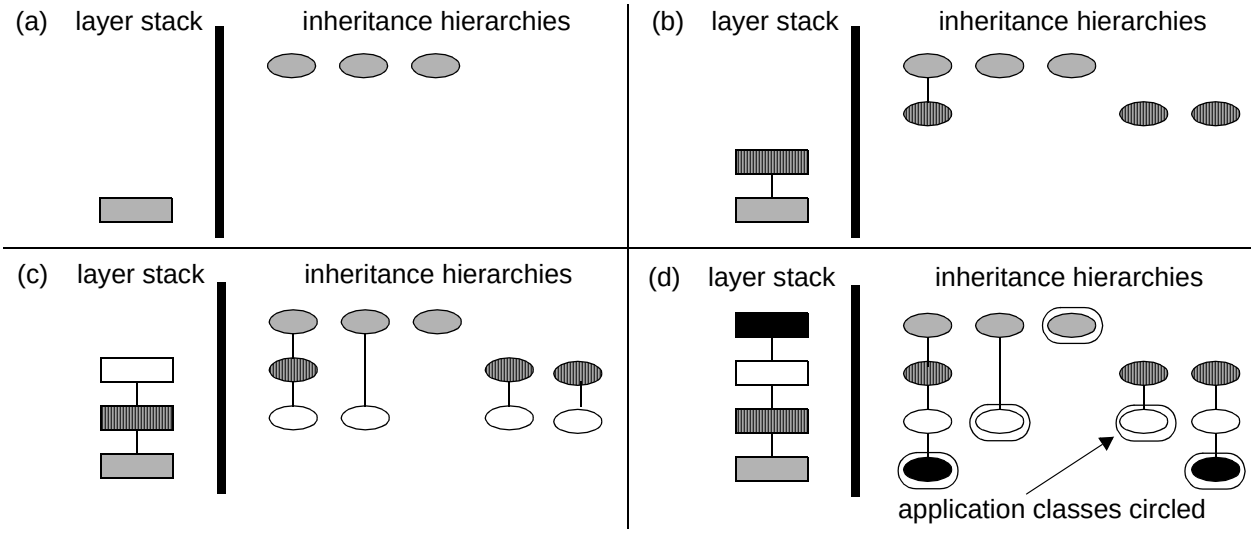

Figure 2: Component Composition and Inheritance Hierarchies

This lead us to develop the *Jakarta Tool Suite (JTS)*, which is a PLA of Java dialects [Bat98]. JTS is, in fact, a GenVoca "generator" by which dialects of Java are assembled by composing symmetric components. Presently, JTS has components that extend Java with the features that include Lisp backquote/comma (to specify and manipulate code fragments), hygienic macros (to avoid the inadvertent capture problem), parameterized inheritance, and a domain-specific language for container data structures. JTS is bootstrapped, so that JTS is written in a dialect of Java that was produced by JTS itself. I invite readers to download beta versions of JTS (in October 1998) at the web site **`http://www.cs.utexas.edu/users/schwartz/`**.

5 Conclusions

Heliocentricity was advanced in 1543, yet 60 years later it had made no impact. One reason was that most people didn't care about retrograde motions. Even open-minded academics, such as Jean Bodin, were skeptical [Boo83]:

> "No one in his senses or imbued with the slightest knowledge of physics will ever think that the earth staggers up and down around its own center and that of the sun… For if the earth moved, we would see cities, fortresses, and mountains thrown down… Arrows shot straight up or stones dropped from towers would not fall perpendicularly, but either ahead or behind…"

How then did heliocentricity take hold? Acceptance was gradual as volumes of evidence from telescopic observations of the heavens accumulated. (The telescope was invented in the early 1600s). Heliocentricity was consistent with other theories, such as those on earth tides. But certainly a contributing factor was its simplicity and elegance in addressing practical scientific problems that were otherwise difficult or impossible to solve.

In this paper, I have tried to motivate future directions of software technology. There is no doubt that product-line architectures, refinements as generalizations of components, and the codeless programming of software plug-and-play will come to pass; the only debatable points are how and when. I have offered GenVoca as a way in which all three can be achieved. Still, it is questionable that GenVoca will take hold. However, there are three reasons to be optimistic. First, there is a considerable amount of experimental evidence for its correctness and value (and I would expect there to be much more in the future). Second, the ideas are constantly being reinvented by others (after all, the idea of plug-compatible components isn't exactly novel and is quite appealing in its simplicity). Third, it addresses a critical need in software: that of reducing complexity. It is well-known that one of the great advantages of OO is its ability to manage and

control complexity through class abstractions. Certainly, anyone who has ever written an OO application understands precisely this point. It is not difficult to recognize that standardizing abstractions of a domain/PLA is a *very* powerful way of managing and controlling the complexity of software in a family of applications. It is this latter point on which the success or failure of GenVoca may rest.

Acknowledgments. I thank Yannis Smaragdakis, Rich Cardone, Lance Tokuda, Scott Page, and Lorenzo Alvisi for their insights in helping me clarify points made in this paper.

6 References

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992, 355-398.

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

[Bat97a] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.

[Bat97b] D. Batory, "Intelligent Components and Software Generators", Software Quality Institute Symposium on Software Reliability, Austin, Texas, April 1997.

[Bat98] D. Batory, G. Chen, E. Robertson, and T. Wang, "Design Wizards and Visual Programming Environments for Generators", *Int. Conference on Software Reuse*, June 1998.

[Bax92] I. Baxter, "Design Maintenance Systems", *CACM* April 1992, 73-89.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Component Reuse", *Int. Conference on Software Reuse*, November 1994.

[Boe96] E. Boerger and I. Durdanovic, "Correctness of Compiling Occam to Transputer Code", The Computer Journal, Vol. 39, No. 1.

[Boo83] D.J. Boorstin, *The Discoverers: A History of Man's Search to Know His World and Himself*, Random House, 1983.

[Boo87] G. Booch, *Software Components with Ada*, Benjamin-Cummings, Menlo Park, CA., 1987.

[Boo93] G. Booch and M. Vilot, "Simplifying The Booch Components", *C++ Report*, June 1993.

[Bro86] M. Broy, B. Moeller, P. Pepper, M. Wirsing, Algebraic Specifications Preserve Program Correctness, *Science of Computer Programming*, Volume 7, 1986, 35-53.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD*, 1993.

[Coh98] S.Cohen and L.M. Northrop, "Object-Oriented Technology and Domain Analysis", *Int. Conference on Software Reuse*, 1998.

[Cza97] K. Czarnecki, U.W. Eisenecker, and P. Steyaert, "Beyond Objects: Generative Programming", *ECOOP'97 Workshop on Aspect-Oriented Programming.*

[D'S98] D. D'Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1998.

[Dij68] E.W. Dijkstra, "The Structure of THE Multiprogramming System", *Communications of ACM*, May 1968, 341-346.

[Edw98] S. Edwards, G. Shakir, M. Sitaraman, B.W. Weide, and J. Hollingsworth, "A Framework for Detecting Interface Violations in Component-Based Software", *Int. Conference on Software Reuse*, June 1998.

[Har93] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", OOPSLA 1993, 411-428.

[Hay98] M.G. Hayden, "The Ensemble System", Ph.D. dissertation, Dept. Computer Science, Cornell, January 1998.

[Hei93] J. Heidemann and G. Popek, “File System Development with Stackable Layers”, *ACM TOCS*, March 1993.

[Hut91] N. Hutchinson and L. Peterson, “The *x*-kernel: An Architecture for Implementing Network Protocols”, *IEEE TSE*, January 1991, 64-76.

[Jac97] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley, 1997.

[Kic97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP 1997.

[McI69] D. McIlroy, “Mass Produced Software Components”, *Software Engineering: Report on a Conference by the Nato Science Committee*, Oct 1968, P. Naur and B. Randell, eds. 138-150.

[Met92] E. Mettala and M.H. Graham, “The Domain-Specific Software Architecture Program”, TR-CMU/SEI-92SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[Nei84] J. Neighbors, “The Draco Approach to Constructing Software from Reusable Components”, *IEEE Trans. Software Engineering*, Sept. 1984, 564-574.

[Obe98] P. Oberndorf, “Community-wide Reuse of Architectural Assets”, in *Software Architecture in Practice*, (Bass, Clements, and Kazman, editors), Addison-Wesley, 1998.

[Ode97] M. Odersky and P. Wadler, “Pizza into Java” Translating Theory into Practice”, ACM POPL 1997.

[Par79] D.L. Parnas, “Designing Software for Ease of Extension and Contraction”, *IEEE Transactions on Software Engineering*, March 1979.

[Par83] H. Partsch and R. Steinbruggen, “Program Transformation Systems”, *Computing Surveys*, March 1983, 199-236.

[Rat98] Rational Software Corporation, “Automating Component-Based Development”, 1998.

[Ren96] R. van Renesse, K.P. Birman and S. Maffeis, “Horus, A Flexible Group Communication System”, Communications of the ACM, April 1996.

[Sin96] V. Singhal, “A Programming Language for Writing Domain-Specific Software System Generators”, Ph.D. dissertation, Dept. Computer Sciences, University of Texas at Austin, 1996.

[Sri96] Y.V. Srinivas and J.L. McDonald, “The Architecture of Specware, a Formal Software Development System” Kestrel Institute Technical Report KES.U.96.7, August 1996.

[Sma98] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. ECOOP, July 1998.

Author:

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712
batory@cs.utexas.edu
512-471-9713