Published in *1996 International Conference on Software Reuse*, Orlando, Florida

Subjectivity and GenVoca Generators

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

Abstract[1]

The tenet of subjectivity is that no single interface can adequately describe any object; interfaces to the same object will vary among different applications. Thus, objects with standardized interfaces seem too brittle a concept to meet the demands of a wide variety of applications. Yet, objects with standardized interfaces is a central idea in domain modeling and software generation. Standard interfaces make objects plug-compatible and interchangeable, and it is this feature that is exploited by generators to synthesize high-performance, domain-specific software systems. Interestingly, generated systems have customized interfaces that can be quite different from the interfaces of their constituent objects.

In this paper, we reconcile this apparent contradiction by showing that the objects (components) in the GenVoca model of software generation are not typical software modules; their interfaces and bodies mutate upon instantiation to a "standard" that is application-dependent.

1 Introduction

It is well-known in photography that there is no single perspective from which all aspects of an object can be viewed; depending on the perspective taken, some aspects may be completely hidden while others appear distorted. This simple observation has relevance to software reuse. Consider an application that models textbooks. A textbook might be an object with attributes **`author_name`**, **`title`**, **`subject`**, **`publisher`**, etc. These would be natural attributes if the application needed to retrieve textbooks on the basis of authorship, content, or title. They would not be appropriate, however, if the application maintained stock and volume information for a warehouse (where authorship and subject are irrelevant), or if the application recorded the materials used in manufacturing textbooks (where subject and title are irrelevant). Clearly, the data and operations that are encapsulated by an object will vary from application to application. This impacts software reuse: objects written for one application may not be reused in another because their "views" or "perspectives" are incompatible, even though both applications deal with the same real-world object.

The principle of *subjectivity* asserts that no single interface can adequately describe any object; objects are described by a family of related interfaces [Har93-94, Oss92-95]. The appropriate interface for an object is application-dependent (or *subjective*). The impact of subjectivity is evident in domain-specific software system generators where the goal is to produce customized software for particular applications. Generators implement models of software domains called *domain models*. A central task in domain modeling is to identify the fundamental objects (or abstractions) of a domain and to define a standardized programming interface for each. However, objects with standardized interfaces seems at odds with the need for generators to produce customized interfaces for the software that they generate.

In this paper, we explore the relationship of subjectivity to a class of generators called GenVoca generators [Bat92]. (We believe that subjectivity impacts *all* generators, but in this paper we focus exclusively on its impact on GenVoca). We show that typical component interfaces (i.e., ones that are cast-in-concrete and that do not change upon instantiation) are far too rigid to be practical; GenVoca components have interfaces that enlarge automatically upon instantiation and hence are subjective (i.e., application-dependent). We review techniques that have been used to achieve subjective interfaces in four independently-conceived generators and present a model that unifies them.

2 GenVoca

GenVoca is a model of software generation ([Bat92, Cog93, Hei93]). Among the tenets of GenVoca is that by standardizing the fundamental abstractions of a domain and their programming interfaces, it is possible to create plug-compatible, interoperable, and

1. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

interchangeable building blocks called *components* or *layers*.[2] The set of GenVoca components that implement the same abstraction/interface is a *realm*. Realm **A**, below, has three components and realm **B** has four.

```
A  =  { x, y, z[B] }
B  =  { m[A], n[A], o[B], p[A,B] }
```

Components are parameterized by the realms/ abstractions that they import, e.g., component **z[B]** implements abstraction **A** (because **z** belongs to realm **A**) in terms of the **B** abstraction (because **z** is parameterized by **B**). Similarly, **p[A,B]** implements abstraction **B** (because **p** belongs to realm **B**) in terms of abstractions **A** and **B** (as **p** is parameterized by **A** and **B**). Components that import the same abstraction that they export are *symmetric* (e.g., **p[A,B]**, **o[B]**).

Component composition is modeled by parameter instantiation. A software system is a named composition of components called a *type equation*. Type equations **s1** and **s2** below define two different systems, both of which export the **B** interface (because their outermost components export **B**):

```
s1 = m[ y ];
s2 = n[ z[ p[ x, m[ x ] ] ] ];
```

Composing components can be interpreted as stacking layers in a hierarchical system. For this reason, we use the terms “component” and “layer” interchangeably.

2.1 The Myth of Standardized Interfaces

GenVoca components are composable because they export and import standardized interfaces. As we noted in Section 1, no single interface captures all views of an object. What then does it mean for a GenVoca interface to be “standardized”? How are operations chosen to be included in an interface? What criteria is used to exclude operations? One could argue if GenVoca generators purport to produce high-performance software for a domain, then *no* operation could be excluded because it might be needed for performance-critical applications. Indeed, when GenVoca interfaces are defined, there are operations that most people would agree are “core” or “intrinsic”, but many other operations are “optional” or “subjective”.

Example. P2 is a GenVoca generator for container data structures [Bat93-94a]. The core operations that one can perform on containers are element retrievals, updates, insertions, and deletions. However, there is an infinite number of optional operations: count the number of elements in the container, return the last element inserted, insert an element after a given element, etc. Core operations are distinguished from optional operations subjectively, i.e., by their perceived need for the target applications that P2 has to support.

The notion that standardized interfaces are immutable or cast-in-concrete in GenVoca is a myth. Each component encapsulates a domain-specific feature. For programmers or other components to take advantage of this feature, it is often necessary for a component to export non-core, component-specific operations. The ability of components to augment the set of core operations that they export and import, of course, destroys any pretense of realm interfaces being immutable or cast-in-concrete. To emphasize this point, it is quite common in GenVoca for the exported interface of a generated system to change with the addition or removal of a component.

Example. P2 has a **size_of** component which maintains a count of the number of elements in a container. This count variable cannot be read by a core operation. Instead, **size_of** exports the nonstandard **read_size** operation to read the count. When **size_of** appears in a type equation that defines a container’s implementation, **read_size** is added to that container’s interface. If **size_of** is removed from the type equation, **read_size** is removed from the interface.

Example. P2 has a **timestamp** component. It appends to every element in a container the time of its insertion. The layer-specific operation **get_timestamp** is added to the cursor class interface for reading element timestamps. If **timestamp** is removed from the container’s type equation, **get_timestamp** disappears from the cursor interface.

The general situation is depicted in Figure 1. Three symmetric layers are shown: each exports and imports the same set of core operations. However, the bottom layer has an extra left operation and the middle layer has an extra right operation. When these layers are composed, all layers are *automatically* extended to support both a left and right operation. That is, the bottom and middle layers modify their realm interface by adding their layer-specific opera-

2. A domain abstraction is often defined by an interrelated network of objects/classes [Jon88, Bat92, Gam94, Oss92-95]. Standardizing the interface of a domain abstraction is accomplished by standardizing the interfaces of its constituent objects/classes.

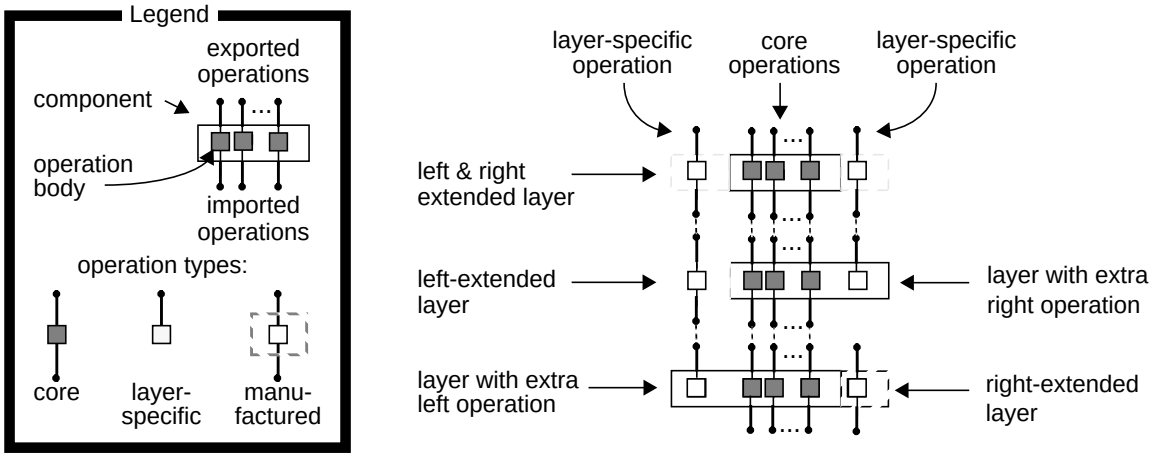

Figure 1. Propagation of Layer Specific Operations

tions. As all layers of a realm export the same interface, every layer of that realm in the type equation must have a left and right operation. By the same reasoning, if the middle or lower layer is removed, its layer-specific operation will be removed from all layers of the composition. It is in this way that GenVoca generators customize the interfaces of components (and their exported objects) and thus produce view-specific software.[3] Furthermore, the ability to add new operations renders the distinction of core v.s. layer-specific operations moot.

However, this does raise an interesting dilemma: on the one hand, the composibility of GenVoca components is dependent on standardized interfaces. On the other hand, individual components may export non-standard operations. Although this seems contradictory, subjectivity offers a resolution.

The principle of subjectivity is that objects have multiple interfaces; the particular interface to be adopted is application-specific. When GenVoca components are composed, their interfaces are automatically adjusted to a standard that is *type equation-specific*. Thus, standard interfaces do not mean cast-in-concrete in GenVoca; they are indeed subjective.

A novel consequence of the above is that, unlike typical software modules, components with subjective interfaces are *freeze-dried* — the set of operations that a component exports *enlarges* upon instantiation. (Which operations are added is type-equation dependent). Thus, a component author never really knows the full interface of his component; an actual interface is only known at instantiation time.

3. Actually, it is unnecessary for the bottom layer of Figure 1 to have a right operation if it is never called. An "dead-code" optimizer would remove such an operation.

Adding new operations to an interface is simple; however, how does one automatically manufacture a method for such operations on a per-component basis? How can components with subjective interfaces be implemented? What programming language features are needed to support subjectivity? What programming paradigm cleanly unifies these ideas? In the following section, we review actual implementations of components with subjective interfaces in four independently-conceived GenVoca generators. Although all four solutions are outwardly different, they are fundamentally similar. Afterward, we distill the essence of these solutions, and in doing so, we answer the questions posed in this paragraph.

2.2 Four Implementations

A hallmark of GenVoca generators is that the design and construction of realm libraries is guided by a careful domain analysis. Components are not ad hoc or randomly harvested modules; they are specifically designed to be interoperable and composable with other components. The constraints on using components in type equations — i.e., their compatibility or incompatibility with other components — is directly encoded as composition rules (a.k.a. *design rules*) in the generator's domain model [Bat95]. However, recognizing composition constraints and adding these constraints to the domain model is the responsibility of domain analysts and component implementors. There is no tool support or automatic way of recognizing the compatibilities and incompatibilities of components; deep domain knowledge is required. From our experience with GenVoca generators, manually recognizing constraints hasn't been difficult since the number of components in GenVoca libraries are rather small (about a few hundred) and experi-

enced domain analysts have no difficulty keeping track of their meaning and distinctions.[4]

Generators perform tasks that are automatable (e.g., code generation, composition, composition validation, optimization, etc.); the tasks that are not automatable (e.g., recognizing new domain abstractions, recognizing new components of a realm, recognizing composition constraints, understanding domain knowledge, etc.) are the responsibilities of domain analysts and component implementors. It is this perspective that one should keep in mind when reviewing the following generator implementations.

Genesis. Genesis was the first GenVoca generator; it demonstrated that customized database management systems (in excess of 50,000 lines of code) could be assembled from prefabricated components [Bat92]. Genesis relied on a rather rigid (and in hindsight) inflexible way of accommodating subjectivity; realm interfaces evolved as new components were written. That is, when a new component **K** was added to realm **R** and **K** exported nonstandard operation **O**, all components of **R** were manually retrofitted to export **O**. This did not mean that every component of **R** had to implement **O**; non-stubbed implementations were provided only for those components where it made sense to do so.

Thus, the interfaces of Genesis components were adjusted manually whenever a new component was added to a realm.[5] There was no subsequent adjustment of interfaces if type equations did (or did not) use a particular component. This approach worked because of the objectives of Genesis, namely, to demonstrate DBMS synthesis. Performance wasn't an issue and a large user community (that would insist on having many optional operations) was not envisioned.

A consequence of this approach was the need for (the above mentioned) design rules: although the interfaces of all components of realm **R** are syntactically identical, not all components implemented operation **O**. This meant that components of **R** were not always interchangeable and that not all syntactically correct compositions of Genesis components were semantically correct. Automatic design rule checking was needed to validate compositions [Bat95].

4. The relatively small size of GenVoca libraries is not a consequence of limited prototypes, but rather the scalability of GenVoca domain models [Bat93, Big94].

5. Components were added to realms in the order that maximally stressed realm interfaces. We discovered that once the first few components were added, realm interfaces quickly reached a steady state. So backtracking and global updating was minimal.

Avoca. Avoca/*x*-kernel demonstrated that highly layered communications protocols could be more efficient and more extensible than monolithic protocols [Hut91, Bat92]. Avoca realm interfaces were rigid (i.e., cast-in-concrete) sets of operations. *Microprotocols*, the name given to Avoca components, implemented a fixed-set of core operations for transmitting messages and opening and closing sessions, plus an additional operation **control**. Every microprotocol could export zero or more control functions — what we have called layer-specific operations — that only it understood. Calls to these functions were made through **control** which took a standard pair of arguments: a control function name and a pointer to the control function's argument list. A **control** operation was implemented as a switch statement; there was one case for each of the microprotocol's control functions and a default case for transmitting the control operation to the next lower microprotocol:

```
void control( int op_id, list *arg_list )
{
   switch( op_id )
   {
   case op1: // code for layer-specific
             // operation #1
   case op2: // code for layer-specific
             // operation #2
   ...
   default : // call control operation
             // of lower layer)
     lower.control( op_id, arg_list );
   }
}
```

The advantage of this approach is its generality; it can accommodate any number of control functions per microprotocol and it does not require component interfaces to be modified (with the addition or removal of a layer-specific operation).[6] The drawbacks are program clarity and performance. Coding function calls via switch statements and marshalling arguments are well-known to be obscure ways of programming [Joh88]. Moreover, there can be a considerable performance overhead in processing control operations. Calling a control function essentially requires polling each component of a type equation to test if it could process the function. Control functions were not called frequently enough in Avoca for their inefficiencies to be problematic.

Ficus. Ficus builds customized file systems from a single realm of components [Hei93]. All Ficus layers

6. Note that a nondefault method, i.e., something other than transmitting the control function call to lower layers, could easily be encoded in this scheme.

support the same set of core operations plus any number of layer-specific operations. The reliance of Ficus on the Unix vnode facility encouraged a uniform treatment of core and layer-specific operations. It also encouraged the interface of a file system to be determined at configuration time, where every layer of its type equation is polled for the set of operations that it implements. The union of all operations from all layers in the file system defines the interface to that file system. All layers of the file system are then automatically extended to support this interface. Since it is not possible to anticipate what operations would be provided by other (possibly yet-to-be-written) layers, every Ficus layer provides a **`bypass`** method for unanticipated operations. Usually, the default method is simply to transmit calls of unanticipated operations to the next lower layer(s). However, nondefault methods do arise.

An example of a nondefault method occurs in protection layers. Protection layers validate access privileges of clients prior to performing file operations. The bypass method for unanticipated operations is to verify the user's ability to access the given file. Variations on this theme (e.g., testing for read-only access or write access) are possible [Hei93-95].

P2. P2 is a generator of container data structures [Bat93-94a]. A P2 layer is a transformation between the export interface of a component and its import interface(s); only layer-specific operations and core operations for which non-identity mappings are performed are defined. When the P2 generator is compiled, the union of the export interfaces of every layer in a realm is determined. Each layer is then automatically extended to support this union interface. Operations that are undefined by a layer are (in effect) supplied default bodies which transmit the operation to the next lower layer. Default methods can be overridden on a per class basis.

A P2 component that has multiple non-default methods is **`monitor`**, which encapsulates the transformation that converts a container into a monitor; i.e., all accesses to the container occur within a critical region. **`monitor`** exports two classes: **`container`** and **`cursor`**. The **`monitor`** rewrite adds a semaphore data member **`sem`** to the **`container`** class and modifies the methods of all **`cursor`** and **`container`** operations by wrapping them with **`wait`** and **`signal`** calls.

Sketches of the **`monitor`** operation rewrites are shown below. **`container_op`** pattern-matches with any container operation and "**`...`**" is bound to its arguments. The rewritten method is enclosed within braces **`{ }`**: a **`wait`** is performed, then the actual operation itself is processed (by the layer immediately beneath **`monitor`**), followed by a **`signal`**:

```
container_op( … )
{  sem.wait();
   lower_container.container_op( … );
   sem.signal();
}
```

The rewrite of cursor operations is different (albeit slightly) from that of container operations: the container semaphore must be accessed indirectly:

```
cursor_op( … )
{  container->sem.wait();
   lower_cursor.cursor_op( … );
   container->sem.signal();
}
```

In general, a bypass method is specified for each class that is exported by a component. It is not too difficult to imagine that even finer granularities of rewrites may be needed.[7]

2.3 A Model of Subjectivity

Although different, there are striking commonalities in the subjectivity mechanisms of the Genesis, Avoca, Ficus, and P2 generators. In this section, we propose a model of these mechanisms as extensions to the P++ language [Sin93, Bat94b]. P++ is a superset of C++ that is specifically designed to support the GenVoca model. Among its extensions are declarations for realms, components, and parameters. The current version of P++ permits the composition of components at compile-time; it does not yet support run-time compositions or the concept of subjectivity discussed in this paper. (Realm interfaces are standardized manually at design-time, much like component interfaces were standardized in Genesis). *Our proposed extensions to P++ have been implemented in the P2 generator, so we will be describing an abstraction of a working system.* Our choice of P++ as the medium of explanation stems from the recognition that language support for a design paradigm greatly simplifies the application and understanding of that paradigm.

As a running example, we will use the container data structure abstraction of P2 [Bat93-94b]. This abstraction is represented by three classes: elements, con-

7. As an example, if an operation only reads a private data member of a class, there should be no need to execute the read within a critical region. Thus the wrapping of wait and signal operations around a method could be selective.

tainers, and cursors. Elements are the objects stored in containers. Cursors are used to retrieve and update objects within containers.

Realms. A realm interface defines a programming interface for a domain abstraction. It is a specification of the prototypes of one or more classes and functions; realms have no variables or data members. The **DS** (container data structures) realm is shown in Figure 2a. **DS** consists of two classes, **container** and **cursor**, that are parameterized by a third class **e**, the class of elements that are to be stored in containers and that are to be accessed by cursors.

To support subjectivity and interface variations, we introduce subrealms to P++, i.e., specializations/subtypes of a realm definition. Figure 2b shows two subrealms of **DS**. **DS_size** extends the **container** class with the **read_size** operation and **DS_time** extends the **cursor** class with the **get_timestamp** operation. Note that the parameter(s) of superrealms are inherited by their subrealms (i.e., **DS** is parameterized by class **e**, thus **e** is a parameter of the **DS** subrealms **DS_size** and **DS_time**). Figure 2c shows an alternative way of defining subrealms as a union of previously declared subrealms.

Components. A P++ component is a *large-scale refinement* of its realm interface. It is defined as a set of consistent data refinements, non-bypass operation refinements, and bypass refinements that help implement the component's realm interface. A specification of the **size_of** component is shown in Figure 3a. **size_of** refines the **container** class by adding the variables **lower** and **count**, and explicitly refining the **read_size** and constructor operations. All other **container** operations are implicitly refined by the **container** bypass. **size_of** refines the **cursor** class by adding the **lower** variable, plus explicit refinements of the **insert** and **remove** operations (that increment and decrement **count**). All other **cursor** operations are implicitly refined by the **cursor** bypass. There are three points about this example that we want to elaborate.

First, rewrites of unspecified operations are expressed by the P++ **bypass** construct. **bypass** pattern-matches with the name of any operation that is not explicitly declared within the enclosing class but is an operation that is to be exported by that class. **bypass_type** is the return type of that operation and **bypass_args** matches its argument list. The body of **bypass** defines the method rewrite. For example, the **size_of** bypasses for both **cursor** and **container** transmit the operation verbatim to the layer immediately beneath **size_of**. Figure 3b shows the **monitor** component which does not use verbatim bypasses.

(a)
```
template <class e>
realm DS
{
   class container
   { container ();
     bool is_full();
     ... // other operations
   };

   class cursor
   { cursor (container *c);
     container *cont();
     void advance ();
     void insert ( e *obj );
     void remove ();
     ... // other operations
   };
};
```

(b)
```
template <class e>
realm DS_size : DS< e >
{
   class container { int read_size(); }
};

template <class e>
realm DS_time : DS< e >
{
   class cursor { int get_timestamp();  }
};
```

(c)
```
template <class e>
realm DS_size_time : DS_size<e>, DS_time<e>;
```

Figure 2. Realm and Subrealm Declarations

Second, bypasses complicate type checking in P++ because they allow interfaces of component instances to be of an arbitrary size. Consequently, component instances can have widely varying realm export and import types. To type check component definitions, all we need to ensure is that the type signatures of the realm operations that are explicitly referenced in the component body match those of the export or import realms. For example, **size_of** explicitly exports the **insert**, **remove**, and **read_size** operations; their signatures are covered by the **DS_size** realm. (These signatures could also be covered by **DS_size_time** and many other larger realms; **DS_size** is the smallest cover given the realms of Figure 2). Further, **size_of** explicitly imports the **insert** and **remove** operations; their signatures are covered by the **DS** realm. Thus, the **size_of** component is declared to minimally export the realm **DS_size<e>** and to minimally import **DS<e>**.

(a)
```
template <class e, DS<e> x>
component size_of: DS_size< e >
{
  class container
  { friend class cursor;
    x::container lower;
    int          count;

    container()    { count = 0; };
    int read_size(){ return count; };

    bypass_type bypass(...)
    { return lower.bypass(...); };
  };

  class cursor
  {  x::cursor lower;

     e* insert( e *element )
     {  cont()->count++;
        return lower.insert(element);  };

     void remove()
     {  cont()->count--;
        lower.remove();  };

     bypass_type bypass(bypass_args)
     { return lower.bypass(bypass_args); };
  };
};
```

(b)
```
template < class e, DS<e> x >
component monitor: DS< e >
{
  class container
  { friend class cursor;
    x::container lower;
    semaphore    sem;

    bypass_type bypass(bypass_args)
    { bypass_type tmp;
      sem.wait();
      tmp = lower.bypass(bypass_args);
      sem.signal();
      return tmp;  }
  };

  class cursor
  { x::cursor lower;

    bypass_type bypass(bypass_args)
    { bypass_type tmp;
      cont()->sem.wait();
      tmp = lower.bypass(bypass_args);
      cont()->sem.signal();
      return tmp;  }
  };
};
```

Figure 3. The size_of and monitor Component

Third, an implicit assumption of the **DS** abstraction is that the only way elements can be added or removed from containers is via the cursor operations **insert** and **remove**. Should a new layer **L** introduce another operation for adding or removing elements, the **size_of** component may not maintain an accurate count of the number of elements in a container. This means that **size_of** cannot be composed with **L** to yield a valid type equation. Such a constraint can be expressed using design rules [Bat95]. Alternatively, **size_of** could be made compatible with **L** if it defines rewrites for the element insertion and deletion operations of **L**. As mentioned in Section 2.2, the recognition of the incompatibility of component compositions (or the modification of components to make them consistent) is borne by domain analysts and component implementors, and is not done automatically by generators.

Type Equations. Components are composed in P++ in **typedef** declarations. Suppose **array** and **avl** are components that implement the **DS** interface and do not export layer-specific operations. Type equations **C1** and **C2** (below) would generate systems that export the **DS_size** interface:

```
typedef size_of[avl]   C1;
typedef size_of[array] C2;
```

Given these declarations, the program of Figure 4 is type correct. An environment variable decides whether container and cursor implementations of type **C1** or **C2** should be used during program execution.

```
main()
{  DS_size::container *cont;
   DS_size::cursor    *curs;

   if (environment_variable)
   {  cont = new C1::container;
      curs = new C1::cursor;   }
   else
   {  cont = new C2::container;
      curs = new C2::cursor;  }
   ...
}
```

Figure 4. Environment-Selectable Implementations

Now suppose **avl** and **array** are modified to export layer-specific operations: **avl** additionally exports the **num_balances** operation, while **array** additionally exports the **num_free_slots** operation. As explained in Section 2.1, the compositions **C1** and **C2** will generate different systems, both of which have slightly different interfaces than **DS_size**. **C1** would

export the **DS** core, **num_balances**, and **read_size** operations, while **C2** would export **DS** core, **read_size**, and **num_free_slots**. Note that the program of Figure 4 would no longer be type correct (as **C1**, **C2**, and **DS_size** are distinct types), and will fail to compile.[8] This, despite the fact that the additional operations that were generated, **num_free_slots** and **num_balances**, are never referenced.

The problem is that **C1** and **C2** have manufactured interfaces that don't match any explicitly defined realm. For an application to insulate itself from irrelevant operations of components, it must use a realm declaration that defines the interface that all generated systems should export. This could be accomplished by *casting* type equations to yield the subjective view that is required:

```
typedef (DS_size) size_of[avl]   C1;
typedef (DS_size) size_of[array] C2;
```

That is, our application interacts with generated subsystems via interface **DS_size**. **C1** and **C2** are now equations that define different systems that implement **DS_size**. Hence, instances of **C1** and **C2** are plug-compatible and thus the program of Figure 4 is now type correct. From the perspective of the P++ compiler, casting may actually simplify the composition of components. Once the export interface of a generated system is known, operations that do not belong to this interface need not be generated.

Open Problems. Our extensions take us closer to a better understanding of programming language support for GenVoca and components with subjective interfaces. However, several important open problems remain. P++ components are presently composable only at application compile-time; ideally, components should also be composable at run-time. Such a capability would permit software systems to evolve dynamically. Although there are several possibilities on how to proceed (e.g., [For94, Hei93, Hut91]), it is not clear what run-time capabilities should be added to P++ to support the dynamic composition of components with bypass methods.

Another challenging problem is how to encapsulate design rules within P++ components. Presently, design rule checking is accomplished with a tool external to P++ [Bat95]. Thus, design rules for components are specified separately from P++ component definitions. The difficulty of integration is that design rules would extend the P++ type checking system, thereby requiring P++ to be a fairly "open" compiler. Once again, there are possibilities on how to proceed (e.g., [Oss95]).

8. Compilation will fail because types **C1** and **C2** do not have identical signatures and are not explicitly related as subtypes of **DS_size**.

3 Related Work

Frameworks. An object-oriented *framework* is a set of abstract classes with their own set of concrete classes. The combinations of concrete classes that can work together can be defined in a variety of ways (e.g., informally or using factory design patterns [Gam94]); there is no fixed rule about how concrete classes can be paired. Realms and frameworks are indeed similar [Bat92]: the *n* classes of a realm's interface correspond to the *n* abstract classes of a framework. Each GenVoca/P++ component specifies an *n*-tuple of concrete classes (one concrete class per abstract class) that work together as a unit. The differences between realms and frameworks are (a) the subjective nature of component interfaces and (b) the need for bypass methods to encapsulate the operation refinements of components.

Subjectivity. Subjectivity arose from the need for simplifying programming abstractions, e.g., defining views that emphasize relevant aspects of objects and that hide irrelevant details [Shi89, Hai90, Gam94]. This lead to a connection of object modeling with view integration in databases [Elm89], namely, objects models can be defined as a result of integrating different application (or sub-application) views of objects [Gol81, Har92]. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be produced automatically by composing "building blocks" called extensions [Oss92]. An *extension* encapsulates a primitive aspect or "view" of a hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy. A customized "view" of an inheritance hierarchy could therefore be defined by composing extensions. Extensions and their compositions are similar to the GenVoca concepts of components and type equations. Moreover, similar scalability arguments have been advanced independently for both models and that not all compositions of extensions (or GenVoca components) may be semantically correct (c.f., [Bat93] and [Oss92]). The models are not the same, however, as (for example) extensions have no counterparts to realms and realm parameters.

It is worth noting that a rather different and powerful approach to views and software reuse has been pro-

posed by Goguen [Gog86], Novak [Nov95], and Van Hilst [Van95]. The essential idea is to define generic "packages" that present a customized interface to an object (or sets of objects). A view defines a mapping of each object to its customized "perspective".

Module Interconnection Languages (MILs). Limited forms of subjectivity can be achieved through MILs. Microsoft's Common Object Model (COM) permits objects to have a set of (upwards compatible) interfaces to maintain backwards compatibility with old views of objects [Mic95]. As another example, Goguen's model of parameterized programming (LIL) permits simple transforms on modules, such as combining modules by merging their operations and types; types, operations, and exceptions can be added, exchanged, removed, or renamed, etc. [Gog86, Tra93]. While the basic transforms are present to achieve subjectivity, there are no higher-order transforms that query module interfaces, wrap all or selected operations of a module, and propagate operations to other modules; such capabilities can only be specified manually on a per module basis.

Reflectivity. Bypass methods correspond to *method wrappers* or *before and after* methods in metaobject protocols [Kic91]. CLOS was among the first languages to have method wrappers. Wrappers in CLOS are different than in P++ as they are defined on a per-operation basis. A model of wrappers that is closer to P++ is that of SOM metaclasses, where all (or selected) operations of a class can be wrapped by before and after methods [For95]. Wrappers are defined in SOM by overriding the dispatch methods of metaclasses. Thus, to define the equivalent of the P++ `monitor` component would require four separate definitions in SOM: two classes (`cursor` and `container`) and two metaclasses (a metaclass for wrapping `cursor` operations and a metaclass for wrapping `container` operations). There is no mechanism in SOM (and CLOS) to encapsulate multiple classes and metaclasses. In contrast, the P++ component construct allows multiple classes to be encapsulated and does not require the need for metaclasses to specify wrappers. Another important distinction is that wrappers are composed in SOM and CLOS through class inheritance; wrappers (bypass methods) are composed in P++ through realm parameter instantiation. Thus, the mechanism for wrapper composition in both models is quite different.

4 Conclusions

Generators are important tools for software development. Understanding their principles is crucial to their technical development and promulgation. In this paper, we have explored an unusual feature of software components that are used by GenVoca generators. Each component encapsulates a primitive feature of a domain and is specified as a large scale program refinement, i.e., a consistent set of data and operation refinements of multiple classes. Unlike traditional software modules whose interfaces remain unchanged upon instantiation, GenVoca components mutate upon instantiation — their interfaces and bodies enlarge automatically to meet interface requirements that are imposed by a system. The mutability of interfaces is interesting in the context of GenVoca because the composibility of components is based on components exporting and importing standardized interfaces.

We have shown that standardized interfaces and mutable interfaces are not inconsistent. The principle of subjectivity asserts that objects do not have single interfaces, but rather are described by a family of related interfaces. At component instantiation time, an interface is manufactured for each object/class of a component that is appropriate to the system in which it is to be used. Thus, all components in a system that export or import these objects/classes must use this system-specific standard. It is in this way that the interfaces of GenVoca components are customized.

We reviewed different techniques that have been used to implement subjective interfaces and have proposed a model, based on the P++ language, that distills the essential linguistic extensions needed for programming language support. The key features are: realm lattices (i.e., subtyping refinements of programming interfaces that are defined by a set of interrelated classes), bypass methods (i.e., wrappers that define default refinements for operations), and the automatic propagation of operations through components that have been composed by realm parameter instantiation.

Our work is only a first step in understanding the phenomena of subjective interfaces. Many exciting and challenging open issues remain: support for runtime compositions, integration of design rule checking with P++, experimentation with the proposed features, development of domain modeling techniques that incorporate interface subjectivity, and formalization of GenVoca concepts [Cha94, Nen95].

Acknowledgments. I thank Reed Little (SEI) for pointing out the similarity of method wrapper mechanisms in CLOS and FLAVORS to the operation bypasses in GenVoca components. I also thank Lance

Tokuda, Vivek Singhal, Trudy Levine, Peter Clark, Jeff Thomas, and Guillermo Perez for their useful comments on earlier drafts of this paper. Finally, I thank Ira Baxter for his helpful comments on revising this paper.

5 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, et al., "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat94a] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT* 1994.

[Bat94b] D. Batory, et al., "The GenVoca Model of Software-System Generators", *IEEE Software*, September 1994.

[Bat95] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators", *ICSR 1996 (this proceedings)*.

[Big94] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *ICSR 1994* (Rio de Janeiro).

[Cha94] C. Chambers and G.T. Leavens, "Type Checking and Modules for Multi-Methods", *OOPSLA 1994*.

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

[Elm89] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings, 1989.

[For95] I.R. Forman, et al. "Composition of Before/ After Metaclasses in SOM", *OOPSLA 1994*.

[Gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gog86] J.A. Goguen, "Reusing and Interconnecting Software Components", *Computer*, February 1986.

[Gol81] P. Goldstein et al., "An Experimental Description-Based Programming Environment: Four Reports", TR CSL-81-3, Xerox PARC, March 1981.

[Hai90] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE TSE*, November 1990.

[Har92] W. Harrison, et al., "Integrating Coarse-grained and Fine-Grained Tool Integration", *Workshop on Computer-Aided Software Engineering*, July 1992.

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

[Har94] W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary", *Addendum to OOPSLA 1994*.

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

[Hei95] J. Heidemann, email correspondence, 1995.

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

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

[Kic91] G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Mic95] Microsoft, "The Component Object Model Specification", March 1995.

[Nen95] M. Nenninger and F. Nickl, "Implementing Data Structures by Composition of Reusable Components: A Formal Approach", *ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, April 1995.

[Nov95] G.S. Novak, "Creation of Views for Reuse of Software with Different Data Representations", *IEEE TOSE*, December 1995.

[Oss92] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies", *OOPSLA 1992*.

[Oss95] H. Ossher, et al., "Subject-Oriented Composition Rules", *OOPSLA 1995*.

[Shi89] J.J. Shilling and P.F. Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm", *OOPSLA 1989*.

[Sin93] V. Singhal and D. Batory, "P++: A Language for Large-Scale Reusable Software Components", *WISR* (Owego, New York), November 1993.

[Sym84] Symbolics, Inc., *Intermediate Lisp Programming*, September 1984.

[Tra93] W. Tracz, "LILEANNA: A Parameterized Programming Language," *ICSR 1993*, Lucca, Italy.

[Van95] M. Van Hilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", Dept. Computer Science and Engineering, University of Washington, TR 95-07-02.