

# Deriving Components from Genericity\*

James Sasitorn  
Rice University  
6100 South Main St.  
Houston TX 77005  
camus@rice.edu

Robert Cartwright  
Rice University  
6100 South Main St.  
Houston TX 77005  
cork@rice.edu

## Abstract

There is a growing recognition that programming platforms should support the decomposition of programs into *components*: *independent* units of compiled code that are explicitly “linked” to form complete programs. This paper describes how to formulate a general component system for a nominally typed object-oriented language supporting *first-class* generic types simply by adding appropriate annotations and syntactic sugar. The fundamental semantic building blocks for constructing, type-checking and manipulating components are provided by the underlying first-class generic type system. To demonstrate the simplicity and utility of this approach to supporting components, we have designed and implemented an extension of Java called Component NEXTGEN (CGEN). CGEN, which is based on the Sun Java 5.0 `javac` compiler, is backward-compatible with existing code and runs on current Java Virtual Machines.

## Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features—*classes, objects, components*

## General Terms

Design, Language

## Keywords

first-class generics, Java implementation, custom class loader, modules, components, signatures

## 1. Introduction

Java has revolutionized mainstream software development by supporting clean object-oriented design, comprehensive

\*This research has been partially supported by the National Science Foundation, the Texas Advanced Technology Program, and Sun Microsystems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'07 March 11-15, 2007, Seoul, Korea  
Copyright 2007 ACM 1-59593-480-4 /07/0003 ...\$5.00.

static type checking, “safe” program execution, and an unprecedented degree of portability. Despite these significant achievements, the Java language has been handicapped as a vehicle for writing large applications by two major shortcomings: (i) the lack of a generic type system (classes and methods parameterized by type) and (ii) the absence of a component system for decomposing applications into independent units with statically checked interfaces. The first shortcoming has been partially addressed by the Java standardization process (JSR-14 [11] and Java 5.0), and more comprehensively by the programming research community (GJ [4], PolyJ[13], NEXTGEN [5], LM [20, 19]) but the issue of how to support a general purpose component system in Java has received scant attention.

Components are independent units of compiled code that can be “linked” to form complete programs. They have no independent state, and all of references that cross unit boundaries must be explicitly identified in a unit’s signature and linked together when the units are joined to form programs[8, 10, 17].

Two features of the Java platform are often cited as mechanism for supporting components: the Java package system and the Java Beans framework. But neither of these facilities qualifies as a component system suitable for decomposing application programs into independent units of compiled code.

Java packages are simply separate name spaces; they typically contain explicit references to names in other packages (akin to hard-coded file names scattered in shell scripts), violating the principle that all external references must be explicitly identified in a component signature. These “imported” references form a web of hidden contextual dependencies,<sup>1</sup> preventing packages from being deployed independently. The only way to alter the references imported by a package is to manually edit the package and recompile it—a tedious error-prone process.

The Java Beans framework is a “wiring standard” including an introspection mechanism for building applications in specific domains. Java Beans is targeted at building graphical user interfaces in client programs. A variant of this framework, called Java Enterprise Beans, focuses on distributed transaction processing systems (“middleware”). Neither of these wiring standards is designed to support the natural decomposition of applications outside these targeted domains. Moreover, like all “wiring standards”, they formulate components as objects (rather than classes) compromising the effectiveness of static type checking and preventing the use of linguistic mechanisms like inheritance across component boundaries.

<sup>1</sup>Imported classes may depend on classes that are not imported.

Developing a general component system for an object-oriented language is a challenging design problem because inheritance across component boundaries can produce unexpected results. The name of a method introduced in a class may collide with the name of a `public` or `protected` method in an imported superclass [18, 17]. This “accidental method capture” problem inhibits using classes as components because a component class often contains methods beyond those required in a particular program.

Component NEXTGEN (CGEN) avoids method capture by supporting “hygienic” components based on “hygienic” mixins [8, 9, 1] that eliminate accidental method capture by systematically renaming methods. CGEN is an extension of the NEXTGEN language, a generalization of Java 5.0 that supports first class generic types [15]. NEXTGEN is backwards compatible with existing Java libraries, Java legacy code, and existing Java Virtual machines. The NEXTGEN compiler is based on the Java 5.0 `javac` compiler but generates different code for generic classes and polymorphic methods. The presence of first class generic types in CGEN reduces the construction of a hygienic component system for Java to adding the appropriate annotations (signatures), syntactic sugar (modules), and the corresponding component-level type checking.

In NEXTGEN, generic types are “first-class” values that can appear in almost any context where conventional types can appear. NEXTGEN supports type casting and `instanceof` operations of parametric type, class constants of naked parametric type (e.g., `T.class`), and `new` operations of parametric and pure parametric class and array types, e.g. `new Vector<T>`, `new T()`, and `new T[]`. CGEN goes even farther by incorporating *hygienic* mixins, class definitions where a type parameter `T` serves as the superclass:

```
class C<T implements I> extends T { ... }
```

This extension of NEXTGEN was explored in MIXGEN[1], a pedagogic language extending a core subset of NEXTGEN that probes the limits of first-class genericity using the NEXTGEN implementation architecture.

The hygienic semantics for mixins prevents accidental method capture when a mixin instantiation `C<A>` overrides a method of the superclass `A` that is not a member of the interface `I` bounding `A`. Mixins provide the critical machinery necessary to support inheritance across component boundaries. The identity of an imported class `T` is unknown to an importing component when it is compiled. The binding of `T` is not determined until the component is linked with other components. Unfortunately, hygienic mixins are not part of the standard set of abstraction mechanisms associated with object-oriented languages. Yet they constitute the critical enabling technology for decomposing object-oriented programs into components.

The primary technical contribution of this paper is a simple, elegant formulation of components that seamlessly extends the Java language enhanced by first-class generic types.<sup>2</sup> CGEN does not require a secondary module language or additional translation steps between compilation and execution. The CGEN component language is a natural extension of the NEXTGEN generic type system that supports the parameterization of packages by type (creating modules) and adds new annotations (package signatures) to the type system. The translation of these new constructs into NEXTGEN code is a simple local syntactic expansion,

<sup>2</sup>We are confident that essentially the same approach would work for C# enhanced by hygienic mixins.

implying that the new constructs are technically syntactic sugar<sup>3</sup>. Like NEXTGEN, CGEN is backwards compatible with existing libraries and executes on current JVMs.

We have developed a prototype implementation of CGEN derived from our NEXTGEN compiler. The CGEN compiler produces code comparable in efficiency and size to NEXTGEN, which is not surprising since CGEN uses the same implementation architecture as NEXTGEN.

The remainder of this paper is organized as follows. Section 2 reviews critical background on subtyping and mixins. Section 3 assesses the current state of software components on the Java platform, motivating the development of CGEN. Section 4 describes the syntax and semantics of the components constructs that CGEN adds to Java. Section 5 discusses the importance of hygiene in object-oriented component systems. Section 6 describes how CGEN is implemented using techniques developed for NEXTGEN and MIXGEN. Finally, Sections 7 and 8 discuss related and future work on component systems for Java and similar languages.

## 2. Background

Before we discuss how components are defined in CGEN, we begin with a brief explanation of some key concepts underlying our approach.

### 2.1 Nominal versus Structural Subtyping

Mainstream object-oriented languages, namely Java, C#, and C++, rely on *nominal* subtyping rather than *structural* subtyping, which has been the focus of most of the technical research on type systems supporting subtyping. In object-oriented languages with nominal subtyping, a class `A` is a subtype of a class `B` iff `A` explicitly inherits from `B`.<sup>4</sup> The programmer explicitly determines the structure of the type hierarchy.

In languages with structural subtyping, a class `A` is a subtype of a class `B` iff `A` includes all of the member names in `B` and the type of each member in `A` is a subtype of the corresponding member in `B`.<sup>5</sup> There is no relationship between subtyping and code inheritance because subtyping between classes depends only on the signatures of the members of types—not on their inheritance relationships.

To illustrate the differences between nominal and structural subtyping, consider the following Java classes:

```
abstract class MultiSet {
    abstract public void insert(Object o);
    abstract public void remove(Object o);
    abstract public boolean contains(Object o);
}

abstract class Set {
    abstract public void insert(Object o);
    abstract public void remove(Object o);
    abstract public boolean contains(Object o);
}
```

Since Java is based on nominally subtyping, no subtyping relationship exists between the classes `MultiSet` and `Set`.

<sup>3</sup>Our actual implementation optimizes the naive translation based purely on syntactic sugar.

<sup>4</sup>We are using the term inheritance more broadly than inheriting code. In our terminology, a Java class inherits from all of its interfaces as well as its superclass.

<sup>5</sup>The subtyping relation on fields is not the obvious one. A field `f` of type `T` is modeled as a *getter* `T getF()` and a *setter* `void setF(T f)`. Methods are subtyped using the standard subtyping relation on functions: co-variant subtyping on output types and contra-variant subtyping on input types of function types. Hence field types must match exactly for a subtyping relationship to hold between classes.

In contrast, suppose Java supported structural subtyping. Then the class `MultiSet` would be a subtype of the class `Set` and vice-versa because the two classes contain the same public methods with identical signatures. Structural subtyping forces subtyping relationships between class types on the basis of matching method signatures even when method contracts conflict.

## 2.2 Mixins

The *mixin* construct abstracts the process of class extension. As stated previously, a mixin definition

```
class C<T implements I> extends T { ... }
```

looks just like a generic class definition except for the fact that the superclass is a type parameter. The primary technical problem involved in formulating mixins as a language construct is defining the meaning of a mixin instantiation `C<A>` when `A` includes a public method `m` with exactly the same name and type as a method *introduced* by the mixin `C`, *i.e.*, a method that does not appear in the bounding interface `I`. In the naive semantics for mixins, `C<A>` *accidentally overrides* the definition of `m` in `A`—breaking the functionality of `A`.

A *hygienic* semantics for mixins eliminates this pathology by ensuring that all of the methods introduced by a mixin (methods other than those in the bounding interface `I`) have *new* names that do not collide with the names of any of the methods in the mixin superclass. CGEN renames every method introduced in a mixin instantiation (or ordinary subclass of the mixin instantiation) by prefixing it with the (mangled) name of the mixin superclass. Since interface-based method dispatches simply use the method names provided by the interface type, a forwarding method must be generated for every renamed method that belongs to an interface. See [1] for a detailed discussion of the semantics and implementation of hygienic mixins in Java.

## 3. Motivation

Let us examine in more detail the support provided by the existing Java platform (Java 5.0) for software components.

### 3.1 Java Packages

In the Java programming model, packages are the primary construct used by programmers to organize a large-scale Java application into manageable units of development.

Packages provide distinct name spaces to organize collections of classes and interfaces. A class is included in a package by adding a `package` declaration at the top of the file and placing the file in the directory associated with the package. A packaged `public` class can then be used outside the package by either: (i) using a fully qualified reference to a class, or (ii) using an `import` statement coupled with a non-qualified reference. In compiled class files, these two forms of usage are equivalent since non-qualified references are replaced with their fully qualified names and the `imports` are discarded.

Figure 1 shows the outline<sup>6</sup> of a recursive decent parser for the Jam language, a simple functional language assigned as a series of programming exercises in the undergraduate course at Rice University on programming languages. The `Parser` resides in the package `jam.parser` and uses classes from

<sup>6</sup>Due to space constraints a full version could not be included. A full version of JAM and other examples are available at <http://japan.cs.rice.edu/nextgen/examples/>

---

```
package jam.parser;
import jam.ast.*;

class Parser {
    Parser() { ... }
    AST parse(String url) throws ParseException { ... }
}
class ParseException implements jam.ast.Exception {
    String msg;
    ParseException(String msg) { ... }
}
```

---

Figure 1: Jam parser definition

---

```
import jam.ast.*;
import jam.parser.*;

class Interpreter {
    public Interpreter() { }
    public JamVal interp(String url) {
        Parser p = new Parser(url);
        try {
            jam.ast.AST tree = p.parse();
            ...
        } catch (ParseException e) { ... }
    }
}
```

---

Figure 2: Jam Interpreter definition

`jam.ast`. Figure 2 shows the outline of an interpreter that references `jam.parser.Parser` and classes from `jam.ast`.

From a software engineering perspective, the fully qualified references to `jam.ast.Exception` in Figure 1 and `jam.parser.Parser` in Figure 2 pose long-term code management problems. Each fully qualified identifier is essentially a hard-coded, absolute reference to an external class. Changing this imported class or its containing package name, requires a global search and replace of all these references scattered throughout the program.

When viewed from a macroscopic level, these references create a tangled web of hidden contextual dependencies across packages. As a result, packages cannot be developed in isolation. All package dependencies must be present for both compilation and execution. Modifications to classes in a package typically require recompilation of all code that depends on it.

These problems become more apparent when we consider alternative component implementations, e.g., a bottom-up parser `jam.botparser.Parser`. Theoretically we should be able to switch to this new parser by changing the import statement in Figure 2 to `import jam.botparser.*`. However, there is no guarantee this alternative parser provides the same interfaces, much less a similar set of classes, until we recompile the `Interpreter`. This violates our goal of components being independently compiled units of code.

We could avoid recompilation by passing a `Parser` to the constructor of the `Interpreter`. In essence, this approach is an idiom for manually representing components as objects.<sup>7</sup> On a limited basis, this idiom can enable a Java class to accept minor changes in its imports without modifications to its source code. But as a scheme for systematically eliminating explicit dependencies, it is unworkable. Since components are objects, component linking only occurs during program execution when a class's imports (represented as objects) are

<sup>7</sup>Note: since components cannot have multiple instances, all such classes must be singletons.

passed as arguments to methods in the class’s client interface. In realistic applications, a class may import dozens of classes, producing method signatures with dozens of parameters, which must be modified when a component’s dependency structure changes. Even our pedagogic **Interpreter** class imports 75 classes. Furthermore, there is no mechanism in Java to prevent the introduction of hidden dependencies (explicit external class references) in a component class which only surface when a client uses a new component configuration.

## 4. CGEN Architecture

A Java component has a natural formulation as a collection of generic classes with a common set of type parameters. All external references within a component class can be made manifest by expressing them as type parameters to the class that are instantiated when the component is linked. Hence, component linking can be reduced to the (type) application of generic classes to class arguments. Of course, an actual component system must include some new syntactic machinery for:

- defining components and the component parameters that they import (dependencies);
- declaring the functionality imported and exported by components;
- linking components together; and
- checking that types of linked components match.

Fortunately, a rich generic type system for classes (including hygienic mixins) provides the critical parts for assembling this machinery. In CGEN, we parameterize components by other components (collections of classes) instead of individual classes, but in the underlying JVM implementation, we can reduce component parameterization to class parameterization.

CGEN is an extension to the NEXTGEN formulation of Generic Java. NEXTGEN is a backward compatible extension of Java 5.0 that supports run-time generic types. In CGEN, components are called *modules* rather than *components* because the name **Component** is extensively used in the Java GUI libraries. CGEN modules generalize Java packages. A *module* is a bundle of classes with a name qualifier (prefix) just like a package, but with two significant differences. First, a *module* explicitly declares the functionality—provided by other modules to be named later—that it imports. Second, the functionality exported by a module is explicitly specified by a *signature* consisting of a set of *class prototypes* which describe the interfaces and the “shapes” of classes provided by the module. Class prototypes contain no code. The set of imported modules is specified using generic parameterization—much like the generic parameterization in Java 5.0. Each import parameter includes an upper bound identifying a signature that the imported module implements. These signature bounds provide the crucial machinery to support the separate compilation of modules.

At the language level, signatures play roughly the same role in the context of modules that Java interfaces play in the context of classes, with three major differences. First, modules and signatures appear in disjoint contexts. A module cannot serve as a signature, while a signature can only be used as the bound for a module parameter or the base signature (in an **extends** clause) for another signature. Second,

CGEN modules and signatures are *second-class*. Signatures are mere annotations that have no runtime representation, while modules are flattened into a collection of conventional Java classes and interfaces. Third, the members of modules and signatures are disjoint: the members of a signature are class and interface *prototypes* while the members of a module are classes and interfaces.

When a module parameter with signature **S** is linked with (bound to) a module **M**, **M** must contain a class or interface **C'** for each class or interface prototype **C** in **S**. A class **C'** matches a class prototype **C** iff **C'** has exactly the same name as **C** and has members matching the prototype members declared in **C**.<sup>8</sup> The subtyping relation between modules and signatures is *nominal*: a module **M** implements a signature **S** only if **M** explicitly declares that it implements **S**. Hence, CGEN modules are nominally subtyped just like Java classes.

Since CGEN modules and signatures have second-class status, they cannot be instantiated at runtime, used as arguments to methods, or used in any object-passing protocol. Signatures are annotations which are used exclusively during the compilation process. Modules exist at runtime only in the restricted sense that the members of the module are conventional Java classes and interfaces that are dynamically loaded at runtime. This restriction on modules and signatures simplifies the semantics and implementation of CGEN, while providing sufficient expressiveness to support a rich component system.

Besides modules and signatures, CGEN adds one other construct to NEXTGEN, the notion of *binding* an identifier to a module instantiation. The **bind** construct provides concise names for signature and module instantiations. In addition, it provides a simple mechanism for defining mutually recursive modules.

### 4.1 Module Signatures

A *signature* is a distinct name space containing a collection of class and interface prototypes that are used to constrain the “shape” of a matching module. More precisely, a signature has the form:

```
signature S<V implements E,..., V implements E>
    extends E, ..., E;
sigMember*
```

where **S** is a fully qualified name for the signature (just like a package declaration in conventional Java); **V** is an identifier serving as a module parameter; **E** is either a fully qualified signature name or a signature instantiation; and *sigMember* is either a class or interface prototype, a **bind** statement, or an **import** statement. The **E** in each **implements** clause bounds the preceding parameter. The **E**s following the **extends** clause in the signature expression define signatures from which **S** inherits, akin to Java interfaces inheriting from other interfaces.

Interface prototypes look like ordinary Java interfaces except that they may include references to imported modules (module parameters). Members of imported modules are denoted using familiar Java dot (“.”) notation. Class prototypes look like ordinary Java classes, except that they may reference module parameters and can only provide method signatures—not actual implementations. In contrast to interface prototypes, class prototypes can include constructors, dynamic and static fields, dynamic and static methods, and

<sup>8</sup>In this context and several others, we use the term *class* to refer to either a Java **class** or **interface**

even inner interface and class prototypes as members. All of the members of a signature have **public** visibility. The visibility of the members of interface and class prototypes can either be **public** or **protected**. **public** visibility is the default.

Since many program components depend on (import) other components, signatures in CGEN may be parameterized by modules just as Java 5.0 interfaces can be parameterized by types. Some types in a class or interface prototype within a signature may depend on types provided by a module. In this case, the signature must be parameterized by that module. The requisite Java types can be extracted from the module parameter using member selection. Each import parameter is denoted by a Java identifier and bounded by a signature indicating what members the imported module must include.

A signature instantiation defines a *ground* (fully instantiated) signature by linking the import parameters of a signature with appropriate modules. A signature instantiation has exactly the same syntactic form as a generic type instantiation:

```
sName<sigExpr, ..., sigExpr>
```

where *sName* is a fully qualified signature name and each *sigExpr* is either an imported signature parameter (which is bound to a module), or a *module* instantiation. Note that all signature expressions are ground (fully instantiated). There is no partial application of unlinked modules to module arguments. Module instantiations are defined in Section 4.3. Figures 3 and 4 show the module signatures *SSyntax* and *SParser* used for a component-based parser for the Jam language. The signature *SParser* imports a module implementing the signature *SSyntax*.

As discussed earlier, a signature declares the visible shape (functionality) of a module. In Java terminology, a module *implements* a signature. A module satisfies its signatures if it contains a class or interface matching each class or interface prototype in the signature. The module may contain extra members. Since a class prototype may be a member of a class prototype, the matching process is recursive. This type-checking process is discussed in more detail in Section 4.4.

Signatures are top-level language constructs that have no representation in the JVM.

## 4.2 Module Definitions

A module is a distinct name space that contains a collection of classes and interfaces and stipulates the signatures implemented by the module. More precisely, a *module* definition has the form:

```
module M<V implements E, ..., V implements E>
  implements S, ..., S;
moduleMember*
```

where *M* is a fully qualified name for the module (just like a package declaration in conventional Java); *V* is an identifier serving as a module parameter; *E* is a fully qualified signature name or a signature instantiation bounding the corresponding parameter; each *S* is a fully qualified signature name or a signature instantiation bounding (exported by) the module; and *moduleMember* is either a class or interface definition, an import statement, or a **bind** statement.

As in *signatures*, module dependencies (imports) are defined using generic parameterization. Figure 5 shows a module *JamParser* that implements the signature *SParser*.

---

```
signature SSyntax;

interface AST { }
interface IBinOp implements AST {
  String toString();
}
class BinOpPlus implements IBinOp {
  public static final BinOpPlus ONLY;
}
...
class Exception {
  String getMessage();
}
```

---

Figure 3: Signature for Jam syntax

---

```
signature SParser<A implements SSyntax>;

class Parser {
  Parser();
  A.AST parse(String url) throws ParseException;
}
class ParseException extends A.Exception {
  String msg;
  ParseException(String msg);
}
```

---

Figure 4: Signature for Jam parser

---

```
module JamParser<A implements SSyntax> implements SParser<A>;

public class Parser extends Object {
  Parser() { ... }
  A.AST parse(String url) throws ParseException { ... }
  String[] getLog() { ... }
}
public class ParseException extends A.Exception {
  String msg;
  ParseException(String msg) { this.msg = msg; }
}
```

---

Figure 5: JamParser Module Definition

The contents of a module are type-checked following Java 5.0 type checking rules where module imports *Vs* are synonymous with their bounding signatures and class and interface prototypes within signatures are treated exactly like ordinary class and interface definitions within packages. Then the class and interface definitions in the module must be checked against the bounds provided by the export signatures using *structural matching* as described in checking module instantiations in Section 4.4.

## 4.3 Module Instantiation

The **bind** construct allows third-party composition of modules. It defines a module instantiation by providing the necessary import parameters to satisfy a module's parametric signature. A module binding has the form:

```
bind sName name = mExpr;
```

where *sName* is a fully qualified signature name or signature instantiation, *name* is an identifier, and *mExpr* is either an imported module parameter (which is bound to a module) or a module instantiation. The *sName* can be used to associate a weaker signature with *name* than the export signature of *mExpr*. In this way, the programmer can expose only the requisite part of a module with a larger export signature.

A module instantiation has the same syntactic form as a signature instantiation:

---

```

bind SSyntax JSyntax = JamSyntax;
bind SParser<JSyntax> JParser = JamParser<JSyntax>;

public class Interpreter {
  public Interpreter() { }
  public JSyntax.JamVal interp(String url) {
    JParser.Parser parser = new JParser.Parser(url);
    try {
      parser.parse(...);
      ...
    } catch (JParser.ParseException e) { ... }
  }
}

```

---

Figure 6: Module Instantiation

$mName\langle mExpr, \dots, mExpr\rangle$

where  $mName$  is a fully qualified module name and each  $mExpr$  is either an imported module parameter (which is bound to module) or a module instantiation. Note that all module expressions are fully instantiated. There is no partial application of unlinked modules to module arguments.

The meaning of a module instantiation is literally the set of classes in the instantiated module. It is equivalent to a package in ordinary Java. Hence, there is no such thing as multiple instances (or copies) of a particular module instantiation.

While bindings syntactically resemble variable declarations, they follow the semantic conventions of class definitions. Bind statements occur at the top-level and the bindings are final and immutable. A `bind` can refer to an uninstantiated or a fully instantiated type. Name resolution for bindings follows the same basic type-checking as class declarations. Distinct `binds` are equivalent if they both refer to the same uninstantiated or instantiated type. From a semantic perspective, the name associated with a type in a `bind` statement is *identical* to the type.

The `bind` construct can be used to decompose the construction of a complex module with deeply nested module applications into a more readable format. It can also be used to write impure programs where the impurity is confined to a short preamble and hence easily managed.

Figure 6 shows a client that links and uses the module *JamParser*.

## 4.4 Type Checking Module Instantiations

Assume that we want to check that a module  $M$  implements signature  $S$ . Each class prototype  $C$  in the signature  $S$  must have a matching class  $C'$  in the module with exactly the same name. For every member  $m$  of the class prototype  $C$ , the matching class  $C'$  must contain a member  $m'$  with the same name as  $m$ , same type signature, and same attributes (`public/protected`, `static/dynamic`, and `final/mutable` attribute).<sup>9</sup>

We require an exact match between the types of matching members which is much more stringent than the usual *co-/contra-variant* matching employed in structural subtyping. In principle, we could relax the constraints on matching by using an appropriate collection of *co-/contra-variant* rules that account for the mutability of fields. But we have rejected this more permissive notion of matching because it would force the class loader to change the type signature used in the byte code for every call on a method requiring liberalized matching. Since these calls could occur in any class that

<sup>9</sup>Several of these issues are moot for interfaces.

---

```

module DebugJamParser<A implements SSyntax, B implements SParser<A>>
  implements SParser<A>;
public class Log { ... }

public class Parser extends B.Parser {
  Parser() { ... }
  A.AST parse(String url) { ... }
  Log getLog() { ... }
}

public class ParseException extends B.ParseException {
  String msg;
  ParseException(String msg) { this.msg = msg; }
}

```

---

Figure 7: Inheritance Across Component Boundaries

imports a class with such a method, the class loader could not load a class  $A$  without reading the constant pools for every class  $C$  imported by  $A$  to determine which method calls on  $C$  must be revised. In addition, Java programmers might find this liberalization confusing because the type signatures for some calls in loaded code (as seen by a debugger) would not match the signatures for those calls in class files.

## 4.5 Module Purity

A module is a well-formed component iff it is *pure*—free of fully qualified references to other modules or classes. Fully qualified references can be generated directly by fully qualified names in CGEN source code or by `import` statements. Ground<sup>10</sup> module instantiations with one exception discussed below, are inherently impure. The only impure modules in a component-based program should be a small collection of lightweight “wrapper” modules that link together pure modules.

## 4.6 Integrating Legacy Packages

Legacy Java packages correspond to degenerate modules that take no input parameters and export all of the `protected` and `public` classes in the package. Module code can only import legacy packages by using an explicit `import` statement. Most legacy packages are impure because they reference classes in other packages. A legacy package is *pure* if the only references in the package are to classes in `java.lang`. Of course, the package `java.lang` is *pure* by definition.

## 5. Module Hygiene

Inheritance across component boundaries requires special attention in the design and implementation of CGEN to ensure classes produced during component linking behave as expected.

### 5.1 Accidental Overriding

Consider the module definition:

```
module M<I1 implements S1 ...> implements S
```

The module import  $I1$  is statically typed to the bound  $S1$ . Code defined in  $M$  can access classes provided by  $S1$  using a qualified identifier.

If a class  $C$  declared in  $M$  extends a class  $D$  provided in the import  $I1$ , there can be unintended interference between two classes for any method not declared in the import bound  $S1.D$ .

<sup>10</sup>Containing no module parameters.

This *accidental overriding* can be detected only at runtime when an instance class `D` is linked during module instantiation. In languages supporting separate class compilation, such as Java and `C#`, we cannot detect all such accidental overrides during compilation because each module is viewed as an independent and isolated unit of code.

Figure 7 defines a module `DebugJamParser` that uses a more sophisticated logging facility for the `Parser` class. In the instantiated module:

```
bind SParser<JamSyntax> JParser2 =  
  DebugJamParser<JamSyntax, JamParser<JamSyntax>>
```

the definition of `getLog()` in `DebugJamParser.Parser` *accidentally overrides* the method defined in its super class `JamParser.Parser`, thus breaking its super class.

The issue of accidental overriding of methods by mixin instantiations has been extensively studied in the context of generic Java in [1]. In this work, accidental overriding in first-class generic Java is systematically avoided by a method renaming and forwarding scheme that the class loader applies to every class as it is loaded by the JVM. CGEN employs exactly the same solution to the accidental overriding problem. Every mixin construction in a CGEN module is translated to a mixin class instantiation that is implemented using the techniques described in [1].

## 6. Implementation Details

From the user’s point of view switching to CGEN requires a minimal adjustment from their current Java model: The command `ngc` replaces `javac`, and `nextgen` replaces `java`. The compilation of modules, just like the compilation of generic classes, generates extra class files for CGEN templates. CGEN supports separate compilation of unique modules, but a specific module must be compiled in its entirety.

Support for components in CGEN builds on the existing NEXTGEN infrastructure supporting first-class genericity consisting of a specialized compiler and classloader. The existing parsing, type-checking and name resolution routines required minor adjustments to support signatures and modules definitions. The CGEN compiler contains additional type checking routines to verify a module definition against its declared signatures.

After type-checking, signatures and modules are “flattened” into regular Java code. The flattening of *signatures* is straightforward. Class-prototypes in a signature are flattened just like `static` inner classes. Signature-level parameterization is pushed to the class-level and the classes are moved to the top-level. The fully qualified name of the signature is used as the package for the collection of class prototypes. Since class prototypes only declare specifications, they are compiled into `abstract` classes containing only method signatures without providing actual code. A *manifest* class file is generated to represent the signature. Java class attributes are used to store the specifications of the interfaces imported and exported by the signature as well as the class prototypes contained in the signature. The generated class files are used for static and dynamic type checking of modules— they are not used for any runtime representation in the JVM.

Modules are similarly “flattened”. Module-level parameterization is pushed to the class-level and the classes in the module are moved to the top-level. Hence, the class files generated for a module are simply the class files for the classes in the module. Moreover, the references in these class files

refer directly to classes—not to modules. The fully qualified name of the module is used as the package for the new top-level classes. A *manifest* file is also generated to store the import and export specifications of the module and a list of the prototypes contained in the module.

After “flattening”, the new top-level classes are compiled into templates which are used at runtime to provide fully heterogeneous representations of each type-instantiated class. A template class file looks exactly like a conventional class file except that strings in the constant pool may contain embedded references to the type parameters for the class. These references specify an index of the form `{0}`, `{1}`, etc, specifying which type parameter binding should be inserted when the class file is instantiated. The CGEN classloader replaces these references with the corresponding type arguments (represented as mangled strings) to generate specific generic class instantiations.

`Bind` declarations are processed like `import` declarations. The local names and their bound types are stored in the type environment and used to resolve identifiers in the code. Each identifier is replaced by its real fully-qualified name.

Runtime support for modules incurs minimal additional overhead over the support for first-class generics because module-level parameterization is represented by generic class parameterization. References to imported modules are resolved like references to generic class type parameters.

Since CGEN instantiates modules heterogeneously, this results in the duplication of module code. However, since CGEN instantiates module classes on demand, code duplication occurs only on the classes referenced during program execution.

## 7. Related Work

Jiazzi [10] is a component definition and linking language for Java. Jiazzi uses a stub generator and a static linker that allow components to be written in the unextended Java language. Jiazzi’s linking facilities use the *open class pattern*—a combination of mixins and “upside-down” mixins—to support the modular addition of new features to a set of variant classes. For each linked component, Jiazzi generates a *fixed-package*, a form of fixed-point representing the linked classes. As a result, the semantics of Jiazzi components—even though that are written in “ordinary Java”—is more complex than their familiar Java syntax suggests.

Using a separate linking language like Jiazzi significantly complicates the development of component code. Before a component file can be compiled, the component’s signature must be declared in a Jiazzi `.sig` file, and translated by the Jiazzi stub generator to produce the stub class files required to support the compilation of the component file by the standard `javac` compiler. The linking of compiled components is specified by a Jiazzi `.unit` file which is fed to the Jiazzi linker along with the `.sig` files and class files for the compiled components.

Scala is a multi-paradigm programming language implemented on top of the JVM. Scala integrates features of object-oriented and functional languages. Scala *modules* are singleton classes, but their imports are not declared, so Scala modules typically contain explicit references to other packages. As a result, Scala modules do not qualify as bona fide components.

SmartJavaMod [2] is an extension to Java that supports mixin modules. Mixin modules are a generalization of mixin classes as introduced in [3]. These mixins are *not* hygienic.

Instead of using signatures to statically type modules, SmartJavaMod relies on type inference to determine the principal typing of a module, inferring type constraints for free type parameters. Uninstantiated modules are compiled into polymorphic bytecode similar to NEXTGEN templates. Inferred type constraints are checked when modules are combined. Accidental overriding can be detected in this process and flagged as an error. Unfortunately, it is not clear how to extend this approach to the full Java 5.0 language. In the presence of static method overloading, modules do not have principal typings. For example, given a mixin module with free type variable `T` including dispatches on a variable `v` of type `T` of the form `v.m("2")` and `v.m(new Integer(5))` in contexts of type `void`, a matching class `T` could have a single method `void m(Object o)` or two distinct methods `void m(String s)` and `void m(Integer i)`. Neither of these class types is an extension of the other. In addition, using free type variables is inconsistent with the spirit of Java 5.0 where the type parameterization of classes is explicitly declared rather than inferred.

ComponentJ [16] is a language extension for Java that views components as service providers. Components import and export methods, but not types. Components in ComponentJ are first-class values implemented using objects, implying that there is no inheritance across components.

Our use of modules signatures containing class prototypes is most closely related to the notion of virtual classes in GBETA [7]. In GBETA, class members may be virtual classes, akin to virtual methods in Java. Virtual classes are class valued attributes of an object that can be overridden and extended in subclasses. An overriding class must support all of the members of the class that it overrides. In GBETA virtual classes are accessed relative to an instance of the enclosing class, which means that such classes must be configured as singletons to model components. In CGEN, a module instance is unique for each distinct tuple of type parameters. Virtual classes support extensibility in code development, but classes containing virtual class members typically contain explicit dependencies on other classes, so they are not true components.

The CGEN component framework has also been influenced by the design of module systems for functional languages, most notably ML. The fundamental difference between ML modules (in all the various formulations) and CGEN modules is that ML modules rely on structural subtyping, while CGEN relies on nominal subtyping. This distinction is important because it means that CGEN seamlessly supports mutual recursion among modules, a common occurrence in software engineering practice that creates formidable technical problems in the context of structural subtyping[14, 6].

## 8. Future Work

While CGEN provides a firm foundation for component-based software development, there are several interesting extensions CGEN that warrant exploration. For example, we would like to support reuse between modules. A module `N` could `extend` a previously defined module `M`:

```
module M<I1 implements S1> extends N implements S
```

A major problem in module inheritance is preventing accidental overrides in the classes defined in `N` and `M`. One option is to require class definitions in `M` to be disjoint from those in `N`. A second option, is to allow `M` to define refinements on classes in `N` provided that the new classes structurally match their superclasses in `N`.

Another potential area of research is to extend the CGEN component system to address the extra-linguistic concerns in *Java Specification Request 277: Java Module System*[12].

## 9. References

- [1] E. Allen, J. Bannet, and R. Cartwright. First-class genericity for Java. In *OOPSLA*, 2003.
- [2] D. Ancona, G. Lagorio, and E. Zucca. Smart modules for Java-like languages. In *7th Intl. Workshop on Formal Techniques for Java-like Programs*, 2005.
- [3] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity, and Multiple Inheritance*. PhD thesis, University of Utah, 1992.
- [4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In C. Chambers, editor, *OOPSLA*, pages 183–200, Vancouver, BC, 1998.
- [5] R. Cartwright and G. L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In C. Chambers, editor, *OOPSLA*, pages 201–215. ACM, 1998.
- [6] K. Cray, R. Harper, and S. Puri. What is a recursive module? In *PLDI*, pages 50–63, New York, NY, USA, 1999. ACM Press.
- [7] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *ACM POPL*, 2006.
- [8] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *SIGPLAN*, pages 236–248, 1998.
- [9] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer’s reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, 1999.
- [10] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *OOPSLA*, 2001.
- [11] S. Microsystems. JSR 14: Adding generic types to the Java Programming Language, 2001.
- [12] S. Microsystems. JSR 277: Java Module System, 2005.
- [13] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *POPL*, 1997.
- [14] C. V. Russo. Recursive structures for standard ML. In *ICFP*, pages 50–61, New York, NY, USA, 2001. ACM Press.
- [15] J. Sasitorn and R. Cartwright. Efficient first-class generics on stock Java virtual machines. In *SAC*, 2006.
- [16] J. C. Seco and L. Caires. A basic model of typed components. *Lecture Notes in Computer Science*, 1850:108–128, 2000.
- [17] C. Szyperski. *Component Software*. Addison-Wesley, 1998.
- [18] C. A. Szyperski. Import is not inheritance: Why we need both: modules and classes. In O. L. Madsen, editor, *ECOOP*, volume 615, pages 19–32, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.
- [19] M. Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *SAC*, pages 610–619, 2001.
- [20] M. Viroli and A. Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, 2000.