The Java 5 Generics Compromise Orthogonality to Keep Compatibility

Atanas Radenski
Chapman University
Orange, CA 92866
radenski@chapman.edu

Jeff Furlong
University of California
Irvine, CA 92697
jfurlong@ics.uci.edu

Vladimir Zanev
Columbus State University
Columbus, GA 31907
zanev_vladimir@colstate.edu

Journal title: The Journal of Systems & Software
Final version published online: 19-SEP-2008
Full bibliographic details: The Journal of Systems & Software 81 (2008), pp. 2069-2078
DOI information: 10.1016/j.jss.2008.04.008
The final version of this article (including full bibliographic details) is now available online at: http://dx.doi.org/10.1016/j.jss.2008.04.008

The Java 5 Generics Compromise Orthogonality to Keep Compatibility

Atanas Radenski
Chapman University
Orange, CA 92866
radenski@chapman.edu

Jeff Furlong
University of California
Irvine, CA 92697
jfurlong@ics.uci.edu

Vladimir Zanev
Columbus State University
Columbus, GA 31907
zanev_vladimir@colstate.edu

Abstract

In response to a long-lasting anticipation by the Java community, version 1.5 of the Java 2 platform - referred to as Java 5 - introduced generic types and methods to the Java language. The Java 5 generics are a significant enhancement to the language expressivity because they allow straightforward composition of new generic classes from existing ones while reducing the need for a plethora of type casts. While the Java 5 generics are expressive, the chosen implementation method, type erasure, has triggered undesirable orthogonality violations. This paper identifies six cases of orthogonality violations in the Java 5 generics and demonstrates how these violations are mandated by the use of type erasure. The paper also compares the Java 5 cases of orthogonality violations to compatible cases in C# 2 and NextGen 2 and analyzes the trade-offs in the three approaches. The conclusion is that Java 5 users face new challenges: a number of generic type expressions are forbidden, while others that are allowed are left unchecked.

Categories and Subject Descriptors

D.3.3 [Language Constructs and Features]: *Polymorphism; Data types and structures*; D.3.m [Miscellaneous]: *Generics*

General Terms

Languages, Design

Keywords

Genericity, orthogonality, type erasure, Java, NextGen, C#

1. Introduction

A language without type parameters can still support generic programming. The first version of Java offered a popular generic library of container classes, in which the

role of a type parameter is taken by the root of the class hierarchy, the *Object* class (Gosling and Henry, 1996). Actual type arguments can be inserted through type casts and type tests. In this way, the language offers a *generic idiom* (Bracha et al., 1998b) that is based on well known and established language features, without the complexity of type parameters. Meanwhile, the Java community had promptly recognized that the use of the generic idiom leads to programs that are uneasy to read and to maintain, and are more likely to fail with runtime errors (Bracha et al., 2001). In addition, the extensive use of type casts that are required by the generic idiom is detrimental to execution speed. Some authors consider the use of the *Object* class as a generic parameter too cumbersome or even insufficient to capture some generic concepts (Agesen et al., 1997). Direct linguistic support for genericity had become one of the most frequently requested extensions to Java.

Several popular languages that support generics, such as C++, Ada, Modula 3, Eiffel, Haskell, and ML, have provided valuable experience of how type parameters can be incorporated in Java; see (Garcia R. et al., 2003) for a comparative study. Adding generics to Java has been extensively explored by several groups (Agesen et al., 1997; Bank J. et al., 1997; Bracha et al. 1998b; Allen and Cartwright, 2002). Some of this research experience has been used by Sun Microsystems in a formal process of adding generics to Java. As a result, version 1.5 of the Java 2 platform, also known as Java 5 or “Tiger”, introduced generic types and methods to the Java language.

The Java 5 generics extend Java with type polymorphism. The Java 5 approach is essentially the same as in the GJ language (Bracha et al., 1998b), a descendant of Pizza (Odersky and Wadler, 1997). In Java 5 type parameters can be used to define generic classes and methods. Type expressions can be used as arguments for type parameters. In contrast to the generic idiom, the use of type parameters allows the compiler to replace some runtime type tests with compile time tests, and to generate some useful type tests and type casts, without engaging the programmer in this activity. When compared to the generic idiom, type parameters seem to make generic code easier to read and to reduce the chance of failing runtime tests. Typical use of type parameters in the generic Java 5 is illustrated in Table 1.

```
abstract class Ring<E> {
   E e;
   protected Ring () { }
    abstract Ring<E> add(Ring<E> other);
    abstract Ring<E> mul(Ring<E> other);
    abstract Ring<E> zero();
    E get()        { return e; }
    void set(E e) { this.e = e; } …
}
```

```
class RingOfInt extends Ring<Integer> {
   RingOfInt add(Ring<Integer> other) {
      e = e + other.get(); return this; }
   RingOfInt mul(Ring<Integer> other) { … }
   RingOfInt zero() {
      e = new Integer(0); return this; }
   RingOfInt() { zero(); } …
}
```

Table 1. Definitions[1] of a generic class *Ring* and its client, class *RingOfInt*.

The syntax of the Java 5 generics is superficially similar to C++ templates. This superficial syntactic similarity is intended to provide a feeling of familiarity with the new language features. However, the similarity with C++ does not run deep, because the two languages use completely different translation patterns for their genericity (Ghosh, 2004).

C++ template instantiation is implemented through a translation process termed *code specialization*. Through code specialization, a C++ compiler only instantiates those portions of generic code that are actually used in a particular C++ program. This means that a C++ class instantiated from a class template may have less executable code that a regular non-generic class in C++. A benefit of code specialization is that generic class members that are not needed in a particular C++ program do not in fact exist in the instantiated class code, opposite to a non-generic class which is present in the executable with all its members no matter whether they are used or not [ISO, 1998]. On the negative side, generic type checking is performed at instantiation time, on the template instance, and not on the template itself. This situation can lead to incomprehensible error messages and this can be problematic, especially when the template developer and the template user are different persons.

---

[1] Operations in *RingOfInt* return values of the newly defined type, *RingOfInt*. The parameters of these operations, however, cannot be of type *RingOfInt*. Such parameters can have the composed *Ring<Integer>* type instead. As noted later, the awkwardness described is due to the lack of covariant rules, or type alias facilities (class synonyms). Either would fix the problem.

In contrast to the C++ templates, Java 5 generics are translated by means of an operation termed *type erasure* that removes type parameters and replaces them when necessary with concrete types, typically class *Object* (Bracha et al., 1998a). The translation performs all possible type checks at compile time and automatically generates appropriate type casts. This process is termed *code sharing*. A benefit from this translation pattern is that it generates a single class file to be used by all generic instances. Roughly speaking, the Java translation pattern converts programs that use the Java 5 generics into programs that use the traditional generic idiom.

We have designed a comprehensive generic composition example in order to evaluate the expressivity of the Java 5 generics. Generic composition is a suitable testbed because it represents complex, deep relationships among generic types. This testbed is also used to evaluate the ease of use of the Java 5 generics and to systematically identify Java 5 orthogonality violations.

This paper only focuses on selected generic Java 5 features that are needed for the discussion of all essential orthogonality violations that stem from type erasure, six of them altogether. To keep the size of this article within reasonable limits, we do not discuss generic Java 5 features that are not essential for the analysis of orthogonality violations, such as single and multiple bounds for the type parameters, wildcards, subtypes and super-types of parameterized types, generic methods, and bridge methods. More importantly, the exclusion of these features allowed us to run and compare the same generic experiments in three different languages: Java 5, NextGen 2, and C#. Other publications analyze the special generic features of Java 5 quite well (Bracha et al., 1998b; Bracha, 2004). In addition, a beneficial overview of various changes in the Java language throughout its evolution is offered in (Kamil, 2003).

We chose Java 5 and C# because they are important mainstream languages, and we chose NextGen 2 because it satisfies the Java genericity goals without sacrificing orthogonality so much as Java 5 does. In addition, a separate related work section reviews a number of other projects that are related to Java 5 generics.

The rest of the paper is organized as follows. Section *Generic Composition* discusses the use of generic class composition, providing a comprehensive example of generic composition. Section *Generics Design: Constraints and Solutions* focuses on the Java 5 generics design and its impact on the Java compatibility and orthogonality. This section also compares the Java 5 genericity to alternative approaches adopted in C# 2 and NextGen 2. Section *Orthogonality Violations*

presents six cases of orthogonality violations in the generic Java 5 and also analyzes these same cases in the context of C# 2 and NextGen 2. Alternative approaches to genericity in Java are presented in section *Related Work*. The paper ends with *Conclusions* that summarize the paper's findings and outlines the paper's original contributions.

2. Generic Composition

2.1. Background

Type parameters have been considered critical for using collection libraries in a flexible, yet safe way (Bracha et al., 1998a). Providing adequate support for parameterized collections has been stated as the first goal for the design of the Java 5 generics, and the core Collections API has been targeted as the most important customer of the Java 5 genericity (Bracha et al., 2001; Naftalin and Wadler, 2006). Sun Microsystems has proclaimed that the Java 5 genericity adds compile-time type safety to its Collections Framework and eliminates the drudgery of casting. The vast majority of published samples of Java generics are various generic collections.

Still, the Java 5 generics can have uses that go beyond generic collections. In this paper, we are particularly interested in using the Java 5 generics to define generic classes through *generic composition*. The generic Java 5 Collections API, being straightforward converted-to-generics version of the traditional Java Collections API, heavily relies on class and object composition, but offers very little insight in the potential of generic composition.

The main purpose of this section is to investigate the Java 5 support for generic composition. We offer an extensive example of Java-based generic composition which points to two Java 5 generic features that are detrimental to readability and simplicity. More important, the same generic composition example is employed in later sections as a *testbed* to reveal Java 5 orthogonality violations. In addition to Java 5, the same generic composition example has been also implemented in C# 2 and NextGen 2. This triple implementation has served as a common comparison base for the Java 5, C# 2, and NextGen 2 genericity approaches. This section does not introduce fancier features of the Java 5 generics that are irrelevant to any of our orthogonality concerns. This intentional omission should not be a problem for the reader because all fancy features of the Java 5 generics have already been discussed by others.

Functional composition allows building new functions from existing ones: given functions *f(y)* and *g(x)*, their composition *f(g(x))* defines a function of *x*. *Generic composition* is intuitively similar to functional composition: given type expressions *P<F>* and *Q<E>* that depend on type parameters *F* and *E* correspondingly, their composition *P<Q<E>>* is a type expression that depends on the parameter *E*[2]. Functional composition can employ an arbitrary nesting level, such as in *h(f(g((x)), y)*, and so can generic composition, as in *R<P<Q<E>>, F>* for example. Type expressions designate types. In Table 1, for instance, the type expression *Ring<E>* is used to designate the return type and the parameter type of methods *add()* and *mul()*.

Type expressions are the fundamental core of generic composition in Java. Type expressions can be built by generic or non-generic type identifiers, such as *Ring* and *Integer*, type parameters, such as *E* in *Ring<E>*, and array types. The scope of a type parameter is its introducing class or interface declaration. The composition rules for type expressions are fairly intuitive. For example, *Ring<Integer, Integer>* is an invalid type expression because the generic class *Ring* introduces exactly one type parameter (see Table 1). A complete specification of the generic Java type system is available by Gosling et al., 2005.

[2] In the context of this publication, type expressions involve only type names and parameters. We do not consider type features that are not essential for the analysis of orthogonality violations, such as bounded type parameters, wildcards, subtypes and super-types of parameterized types.

2.2. Generic Composition Example

Some researchers believe that it is particularly difficult to build generic classes by means of generic composition. It is claimed (1) that doing so would require considerable foresight to provide the appropriate parameters, and (2) that it may be problematic to obtain the required subclass relationship (Palsberg and Schwartzbach, 1994). Our experience, however, is that generic composition in Java 5 can be successfully used to construct sets of interrelated generic classes.

Table 2 outlines a sample set of classes declared with extensive use of generic composition. Please, refer to the Appendix for implementations of all classes from Table 2.

Declared Class	Base Class	Recursive Composition
Ring<E>	Object	-
RingOfInt	Ring<Integer>	-
Array<E>	Object	-
ArrayOfRing<E>	Array<Ring<E>>	-
ArrayOfArray<E>	Array<Array<E>>	**Yes**
Matrix<E>	Ring<ArrayOfArray<Ring<E>>>	**Yes**
MatrixOfInt	Matrix<Integer>	-
MatrixOfMatrix<E>	Matrix <Matrix<E>>	**Yes**
RingOfBool	Ring<Boolean>	-
MatrixOfBool	Matrix<RingOfBool>	-

Table 2. Classes declared with extensive use of generic composition.

2.3. Java 5 Difficulties with Generic Composition

Although the Java 5 generics allow for deep generic composition with many levels of generic type nesting, there is more to be desired with respect to readability and simplicity. In particular, the absence of (1) covariant parameters and (2) class synonyms can be detrimental for the simplicity and readability of generically composed classes.

Covariant Parameters. A language supports *covariance* if, when an inherited method is redefined, parameter types or the result type can be replaced by more specific types. Java 5 introduced covariant return types. Consider the following excerpt from our generic composition example (see Table 2 and the Appendix for details).

```
class Matrix<E> extends Ring<ArrayOfArray<Ring<E>>> { …
        Matrix<E> add(Ring<ArrayOfArray<Ring<E>>> other)    {… } …
}
```

The return type covariance of Java 5 allows the return type *Matrix<E>* of the overriding method *add()* in class *Matrix<E>* to be an extension of the return type *Ring<E>* of the overridden method in class *Ring<E>*.

In Java 5, covariance only applies to return types but does not apply to formal parameter types in overriding methods. This limitation forces the use of bulky parameter type expressions such as *Ring<ArrayOfArray<Ring<E>>>other*. While it is unfortunate that covariance does not apply to argument types, it is fair to say that this is not a peculiar drawback of Java 5. In fact, covariant argument types in languages like Java 5 and C# would be statically unsound; hence their implementation would impose some runtime overhead.

Class Synonyms. In a generic language, it can be beneficial to have a class declaration method that is alternative to subclassing (Palsberg and Schwartzbach, 1994). Such an alternative class declaration method introduces a *synonym* of a class, rather than a true subclass, and the synonym can be used where the original class is expected. The original class and its synonym share the same super and subclasses. If the generic Java 5 had this type of mechanism, *Matrix<E>* could have been declared as a synonym of *Ring<ArrayOfArray<Ring<E>>>,* as shown below.

```
class Matrix<E> is Ring<ArrayOfArray<Ring<E>>> { // synonym declaration
        Matrix<E> add(Matrix<E> other){… }  // synonym as parameter type
…}
```

The above example shows how synonyms would have simplified generic definitions and would have made them more readable.

Synonym declarations should be used when subclassing is not actually needed. As a matter of fact, type parameters in the generic Java 5 are effectively synonyms of actual classes, typically class *Object*. Other languages, such as C++ and Standard ML support synonyms, which reduces the verbosity of generic code (Garcia et al., 2003). Unfortunately, synonyms are not universally supported in Java. Introducing synonym declarations in the generic Java 5 would have reduced the complexity of the otherwise bulky type expressions.

3. Generics Design: Constraints and Solutions

In this section, we offer a comparative review of the design of generics in Java 5, C# 2, and NextGen 2 with a focus on design constraints and on adopted solutions. This overview lays the foundation for the next section, where we present six different cases of orthogonality violations in Java 5 and analyze the same cases in the context of C# 2 and NextGen 2.

3.1. Generic Java 5

The design of the Java 5 generics was guided by a comprehensive list of design constraints (Bracha et al., 2001; Cabana et al., 2004). Several crucial design constraints require *compatibility* of the Java 5 generic code with legacy code. These constraints mandate the possibility to:

- Run new generic Java code on an unmodified legacy JVM
- Use legacy Java code in new generic Java code
- Use new generic Java code in legacy Java code

Compliance with the above compatibility constraints ensures, for example, that an applet that uses generics will run on a browser with a standard JVM, including a JVM that has been released prior to the introduction of generics. It should be obvious that compatibility constraints are of paramount importance for a Java language extension, as they are for the extension of any language with a significant amount of legacy software.

Compatibility constraints have been satisfied in Java through the introduction of several new techniques, such as type erasure, retrofitting, raw types, weakened assignment compatibility rules, and unchecked warnings. We introduce these techniques in the rest of this subsection as a preparation for our technical discussion of orthogonality violations.

Type erasure is an operation that transforms Java 5 generic code into code that utilizes the legacy *generic idiom* (see Section 1). When used as a translation pattern, type erasure transforms all instances of a generic class into the same compiled class and this single class exists at runtime to represent all instances.

For example, type erasure translates *Ring<Integer>, Ring<Boolean>,* and *Ring<ArrayOfArray<Ring<Integer>>>* into a single *Ring* class that is used at runtime to implement all of them. The term type erasure is somewhat misleading because it implies erasing all generic type information. In fact, generic type information is stored by the compiler in the class file in the form of an extra

"signature" attribute, an option that is supported by the original JVM class file format. The "signature" attribute is ignored by the JVM, but is now used by the compiler to support separate compilation.

The JVM has never been meant to support type parameters. Hence, generic type information is not loaded into the JVM at runtime (Cabana et al., 2004). This lack of generic type information inhibits runtime type tests and type casts on parameterized types. Attempts to use *instanceof* type tests on type expressions result in compilation error messages. Type casts result in warning messages at compile time and in possible exceptions at runtime, such as *ClassCastException* for example. On the positive side, since the JVM is unchanged in the transition to the generic Java 5, no performance degradation is expected (Gosling et al., 2005).

Retrofitting is a compiler mode that allows adding generic type information to non-generic legacy code. Technically, the developer provides a retrofitting file that defines a new generic interface for the legacy code, and the compiler adds this interface to the class file (using the "signature" attribute discussed above). Thus, a retrofitted legacy file looks as if it were a compiled Java 5 generic file.

To facilitate interfacing with non-generic legacy code, it is possible to use as a legitimate type the erasure of a generic type expression. Such a type is called a *raw type* (Bracha et al., 1998a). A raw type is non-generic and has no type parameters. Special *weakened assignment compatibility rules* allow assignments between generic types and their corresponding raw types. For example, a variable *rawRing* that has been declared to have the raw type *Ring* is assignment compatible with variables *intRing* of type *Ring<Integer>* and *boolRing* of *type Ring<Boolean>.* The assignments *rawRing = intRing* and *rawRing = boolRing* are harmless and are permitted. To ensure compatibility with legacy code, the assignments *intRing = rawRing* and *boolRing = rawRing* are also permitted - despite of them being unsafe and able to contribute to a runtime inconsistency. For example, such assignments allow interoperation between a legacy *Ring* that is based on the generic idiom, and a generic *Ring*.

In order to support interoperability between new generic code and legacy code, the Java 5 compiler allows unsafe raw type assignments, but issues the so-called *unchecked conversion warnings*. These warnings signal the compilation of unsafe assignments that can not be type checked - neither at compile time nor at run time. Such unsafe assignments may produce type inconsistencies at runtime and fail. On the positive end, any source code that compiles without warning messages is

guaranteed to be type safe (Gosling et al., 2005). Type safety in this context means that a *ClassCastException* may only be raised by an explicit cast, but not by an implicit cast added by the type erasure transformation.

3.2. Generic C# 2

The .NET Framework is similar to the Java Platform. It is based on an Intermediate Language (IL), which is analogous to the Java byte-code. Programs are executed by the Common Language Runtime (CLR), the .NET version of the JVM. At the time of our generic experiments, the latest .NET release was 2.0 Beta 2. Through this paper we use short references, such as .NET 2 and C# 2.

When it comes to generics there are considerable architectural differences between the Java 5 Platform and the .NET Framework 2 (Hejlsberg, 2004). While the Java 5 generics were founded on the approach taken in the *GJ language* (Bracha et al., 1998b), the .NET 2 generics were inspired by *Baby IL with generics* (Yu et al., 2004). Each design supports generics, but achieves this new functionality with a different central goal. Java 5 requires full backward compatibility with existing JVMs. In contrast to Java 5, .NET 2 dismisses any backward compatibility requirements with 1.x virtual machines, and strives for forward compatibility only, namely having 1.x code operate successfully on 2.0 platforms.

In .NET 2, the CLR and IL components have been redesigned to allow for generic types to be explicitly represented at runtime. Backward compatibility is lost as a result from the redesign.

Generics are implemented in C# 2 through a combination of code specialization and code sharing.

Generic *value types*, such as *List<bool>*, are implemented via code specialization. This method involves code replication for each value type, a process that is managed by the JIT compiler.

Generic *reference types*, such as *List<string>*, are implemented via code sharing. As discussed in Section 1, code sharing generates a single class file to be used by all generic instances. In addition, the compiler builds type dictionaries that are used by the CLR at runtime to guarantee type-safe use of a shared generic class files.

By using code sharing when possible - instead of pure code specialization – C# 2 limits generic code size overhead to 10% to 20% (Kennedy and Syme, 2001).

An essential feature of the C# 2 generics is the availability of complete type information at runtime (Kennedy and Syme, 2001). This information permits adequate type inference and supports type-safe operations. Also, considerably more type-checks can be performed statically by the C# 2 compiler than by the Java 5 compiler, because of the C# 2 designers' decision not to support raw types. On the negative side, the generic .NET CLR is not backward compatible and may not execute old non-generic code, while a current JVM can execute both types of code. This difference stems from the different goals of the Java and .NET design teams.

We successfully implemented our generic composition example in C# 2 but faced some difficulties. A most significant limitation of the C# 2 generics turned out to be the complete absence of support for covariant types[3]. The C# 2 designers accepted this limitation to increase execution speed (Kennedy and Syme, 2001). This tradeoff forces the developer to use even more verbose signatures than those in Java 5. An illustration of a verbose C# 2 signature is presented below.

```
public override Ring<ArrayOfArray<Ring<E>>> add(Ring<ArrayOfArray<Ring<E>>> other) {…}
```

In contrast to C# 2, Java 5 at least supports covariant return types. An illustration of a less verbose Java 5 signature is presented below.

```
public Matrix<E> add(Ring<ArrayOfArray<Ring<E>>> other){… }
```

We have already criticized (in section 2) the lack of class synonyms in Java 5. Unfortunately, synonyms are not supported by C# 2 as well.

3.3. Generic NextGen 2

NextGen (Allen et al., 2002) is an extension to the GJ compiler that is compatible with existing JVMs. It resembles the .NET approach described in the previous subsection. Like Java 5, NextGen adds generics to the Java language but in contrast to Java 5, NextGen makes generic types available at runtime. Instead of erasing type information, the NextGen compiler keeps type attributes (Allen and Cartwright, 2004): the generic type information is stored in a new template class' constant pool. This pool is used by a custom class loader to create type-safe generic instances. The

[3] Strictly speaking, this is not an orthogonal issue to the point of this paper, because it actually hinders writing both parametric and nonparametric abstractions.

performance of NextGen is similar to that of Java 5. Because some of the operations are in-lined and the average generic instruction overhead is small, NextGen pays little performance penalties. In some cases type safety can be statically ensured by the compiler thus reducing runtime tests. In such cases the NextGen performance improves in comparison to Java 5 (Allen et al., 2002).

Like Java 5, NextGen does not support type synonyms; it also offers covariance in method return types. Covariance for type parameters is announced to be future work for the NextGen project.

We tested our generic composition example (see Section 2) in NextGen 2 - a recent release of NextGen. No code modifications were necessary for the transition from Java 5 to NextGen 2 because our generic composition example employs only generic features that are present in both Java 5 and NextGen 2. Unfortunately, the NextGen 2 implementation was undocumented and unstable at the time of our experiments. Our generic composition example compiled successfully in NextGen 2 but with an unexpected "unchecked cast" warning. Even worse, when the code produced by NextGen 2 was executed in the standard JVM, an unexpected *ClassCastException* was thrown. Our NextGen 2 generic composition experiments were difficult because of the unstable state of the NextGen 2 implementation.

3.4. Tradeoffs in Approaches to Genericity

There are advantages and disadvantages to the three approaches to genericity in Java 5, C# 2, and NextGen 2.

Java's type erasure allows for backward compatibility, but prohibits type information at runtime. This approach is justified for a language with significant legacy software. Unfortunately, when a developer adds generic Java 5 code to non-generic legacy Java code, the reliability of the entire code may become problematic because of the absence of runtime tests in generic code.

By changing the runtime environment, as in the case of C# 2's CLR, generic types can be made available at runtime, but this eliminates backward compatibility. This approach can be acceptable for a language without considerable legacy code. The benefit is that new code can easily be deemed type safe.

Enhanced class loaders, as in NextGen 2, can distinguish types; this otherwise beneficial approach does not seem as well tested as Java's type erasure. Unfortunately, the NextGen 2 developer is expected to implement explicit conversion

routines between generic NextGen 2 code and legacy Java code, to support the strict type rules of NextGen (Allen et al., 2002).

4. Orthogonality Violations

4.1. Overview

The need to satisfy compatibility constraints has led to various orthogonality violations in the generic Java 5. Orthogonality in a programming language means that "a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language" (Sebesta, 2002). "Furthermore, every possible combination of primitives is legal and meaningful." Good orthogonality means unrestricted combinations of primitive linguistic features; poor orthogonality means excessive restrictions on how primitives are used in programs.

The very implementation techniques that have been used to satisfy important compatibility constraints have led to orthogonality violations in the generic Java 5. The choice of type erasure as a translation pattern has been crucial in this respect: It means that type parameters do not exist at runtime. Consequently, a number of language features cannot work well with parameterized types. Type erasure has been detrimental to the possibility of combining parameterized types with such language features as:

- Subclassing
- Constructor invocations
- Class-wide static entities
- Type tests
- Type casts

In some cases, Java 5 disallows the use of features from the above list in parameterized types. In other cases, the application of such features is allowed but the compiler issues an *unchecked warning*, thus telling the programmer that the generated code is unsafe and may fail at runtime.

To conduct orthogonality violation experiments, we have ported our generic composition example (Section 2) to C# 2. Then we have compiled the generic composition example using the following language processors:

- Generic Java:
  - The Sun Microsystems JDK 5
  - The CodeGuide 7 IDE of Omnicore, which was the first IDE to support the Java 5 generics and sometimes provides more meaningful error messages than the JDK (Omnicore Software, 2004)
  - NextGen 2 release 20050221-1718
- Generic C#:
  - C# 2 under the .NET Framework 2.0 Beta 2 release

The rest of this section specifies various cases of orthogonality violations in the generic Java 5 and also presents these same cases in the context of C# 2 and NexGen 2. Each case is illustrated by a brief Java source code sample followed by corresponding error, warning, or information messages for all three languages. A C# 2 source is given only if the code differs from the provided Java code. A NextGen 2 source is not given, as the source is always the same as the Java source.

4.2. Subclassing of a Type Parameter

Subclassing of a type parameter *E*, as in *class T<E> extends E {}* is disallowed in Java 5 (see illustration in Table 3). As a result of type erasure, a type parameter does not have its own runtime representation; hence it cannot serve as a base class. Subclassing of type parameters is disallowed in C# 2 as well but is supported by NextGen 2.

Java 5 source:	class T<E> extends E {}
C# 2 source:	class T<E>:E
JDK 5 message:	*unexpected type found: type parameter E required class*
CodeGuide message:	*This type cannot be subclassed*
C# 2 message:	*Cannot derive from 'E' because it is a type parameter*
NextGen 2 message:	*Compiles successfully*

Table 3. Orthogonality violation illustration: Subclassing of a type parameter

Though the subclassing of a type parameter is forbidden in Java 5, the use of *class T<E> extends Q<E> { }* is allowed in the same language. Indeed, with the use of type erasure, there is no reason to ban the extension of *Q<E>*, as it in fact translates to *T extends Q.*

The subclassing in the form *T<E>* extends *E* of a type parameter E is a desirable feature for the implementation of mixins, because mixins encourage code reuse without the linguistic complexities of built-in multiple inheritance. In contrast to Java 5, mixins are available in Java extensions that run on existing JVMs, such as Jam (Ancona et al., 2000). In NextGen 2, support for mixins is also available.

4.3. Constructor Invocation with a Type Parameter

Creating an instance of a type variable is illegal in all three languages. In Java, this kind of constructor invocation is disallowed because the type is completely unavailable at runtime (see illustration in Table 4). In C# 2, the compiler does not have enough information about the type to generate a working constructor. As can be seen in the error message, the compiler cannot determine if the type defines the *new()* constraint. Recall, all value types include a default no-arg constructor, but reference types do not. Hence, E must be constrained to include the no-arg constructor. In NextGen 2, because type parameters are translated as abstract classes or interfaces, they cannot be directly instantiated. It is indicated that this functionality can be added to NextGen in the future (Allen et al., 2002).

Java 5 source:	E e1 = new E();
JDK 5 message:	*unexpected type found: type parameter E required: class*
CodeGuide message:	*There is no applicable constructor*
C# 2 message:	*Cannot create an instance of the variable type 'E' because it doesn't have the new() constraint*
NextGen 2 message:	*unexpected type found: type parameter E required: class*

Table 4. Orthogonality violation illustration: Constructor invocation with a type parameter

4.4. Constructor Invocation with a Generic Array

It is illegal in Java 5 to instantiate an array whose elements are specified by means of a type parameter in particular and by means of any type expression in general (see illustration in Table 5). Since type erasure drops type information, the JVM has no information as to what type of array object to create (Gosling et al., 2005). In C# 2, adequate type information is available to the CLR hence this kind of constructor invocation is permitted. The NextGen 2 restriction on type parameter constructor

invocation (as discussed in the previous section) is lifted in this case, because no formal constructor is actually being called.

Java 5 source:	E[] a1 = new E[10];
JDK 5 message:	*generic array creation*
CodeGuide message:	*The array creation is not allowed because the created array is actually of type java.lang.Object*
C# 2 message:	*Compiles successfully*
NextGen 2 message:	*Compiles successfully*

Table 5. Orthogonality violation illustration: Constructor invocation with an array type parameter

4.5. Static Access

Using a type parameter to declare a static variable is forbidden in Java 5 (see illustration in Table 6). The Java Language Specification declares that doing so creates a compile time error, although no explanation is given (Gosling et al., 2005). In fact, the absence of type information at run time would make a static generic variable *type unsafe*, as explained later in this section. Because C# 2 does store type information, generic static variable declarations are permitted. Although the NextGen 2 approach claims that any static variables are allowed (Allen et al., 2002), there seems to be problems with their actual implementation, as illustrated by our Table 6 data.

Java 5 Source #1:	static E e2;
Java 5 Source #2:	static E[] a2;
JDK 5.0 message:	*non-static class E cannot be referenced from a static context*
CodeGuide message:	*This parameter type cannot be referenced from static context*
C# 2 message:	*Compiles successfully*
NextGen 2 message:	*non-static class E cannot be referenced from a static context*

Table 6. Orthogonality violation illustration: Static access

Let us assume for a moment that it is possible to declare a static generic variable, *static E e2,* in a generic Java 5 class *T<E>*. Instances of *T<E>* can be defined by substitution of various type arguments for the type parameter *E* of *T<E>* such as for example *T<Object>* and *T<String>*. As already discussed in the Introduction, the Java 5 compiler uses type erasure to generate a single class file for

T<E>. This class file is shared by all generic instances, such as *T<Object>* and *T<String>*. Recall that a static variable such as *e2* is shared between all instances of the generic class *T<E>.* The problem is that the type of *e2* in *T<Object>* is *Object*, while the type of this same variable, e2, is *String* in *T<String>.* To avoid this problem, Java 5 forbids generic static variables. In contrast to Java 5, C# preserves type information at run time. This feature makes it possible to interpret different generic instances of *T<E>* as different types, such as *T<Object>* and *T<String>,* despite the fact that such instances share the same code as compiled from *T<E>*.

4.6. Type Test against Type Expression

It is forbidden to test a Java 5 object against a type parameter, *E* (see illustration in Table 7). As a consequence of type erasure, the type parameter *E* does not exist at runtime, hence the type test *object instanceof E* cannot be supported by the JVM. Consequently, Java 5 programmers need to simulate the missing functionality through alternative and possibly awkward techniques, such as the usage of additional interfaces as wrappers. The Java 5 ban of generic type tests is reduced to unchecked warnings in NextGen 2. (Technically, we have managed to replace error messages with warnings by using the–*Xlint:unchecked* compilation flag.) In C# 2, the type does exist at runtime and can be used for type tests.

Java 5 source:	boolean test(Object o) { return o instanceof E; }
C# 2 source:	bool test(Object o) { return o is E; }
JDK 5 message:	*unexpected type found: E required: class or array*
CodeGuide message:	*Warning: It is not possible to check at runtime whether an instance of type java.lang.Object is of type E*
C# 2 message:	*Compiles successfully*
NextGen 2 message:	*warning: [unchecked] unchecked cast found: java.lang.Object required: E*

Table 7. Orthogonality violation illustration: Type test of a type parameter

Not only is it forbidden to test a Java 5 object against a type parameter, *E*, but it is also forbidden to test a Java 5 object against a type expression, such as *T<E>*. The only difference between the two cases is in the Java 5 implementation that produces different error messages. Type erasure is the common cause of both restrictions.

4.7. Type Casts

In Java 5, using a type expression, *T<E>,* to cast an object is not formally forbidden because of the need to support amalgamation of suitable non-generic legacy code with new generic code. Although some type casts can be statically checked, most casts require run-time checks. However, in the absence of runtime generic type information, generic typecasts cannot be verified at run time. This is why the Java 5 compiler generates a warning message, indicating that the cast is actually not checked at run time (Gosling et al., 2005). Such unchecked casts can easily trigger run time type cast failures in completely unexpected places of the program. In C# 2, since the type information is available, the cast is valid. NextGen 2 implements T as an abstract class. T<E> is implemented as a subclass of T, and the implementation is called an instantiation class. NextGen 2 uses several implementation techniques for type casts, depending on the code context. In the simplest case, the instantiation class of T<E> is used to type-cast. When subclassing is involved, the so-called instantiation interface for T<E> is used; however, this type must be recast to the base type of T (Allen et al., 2002). In general, NextGen 2 can not safely implement all possible type casts, and the NextGen 2 compiler issues warnings like the Java 5 compiler.

Java 5 source:	T<E> method() { return (T<E>) new Object(); }
JDK 5 message:	*warning: [unchecked] unchecked cast found: java.lang.Object required: T<E>*
CodeGuide message:	*Warning: This cast is unsafe because it is impossible to check at run time whether an instance of type java.lang.Object is of type T<E>*
C# 2 message:	*Compiles successfully*
NextGen 2 message:	*warning: [unchecked] unchecked cast found: java.lang.Object required: T<E>*

Table 8. Orthogonality violation illustration: Type casts

4.8. Orthogonality Tradeoffs

Our study of orthogonality violations in the generic Java 5 has led us to the following conclusions:

- The use of parameterized type expressions to declare variables, method return types, and method parameter types is straightforward and unrestricted.
- All other references to parameterized type expressions are problematic – they are either disallowed, or they are allowed but unchecked.

We have found out that the main cause for the orthogonality violations in Java 5 is the type erasure translation pattern - which has been adopted as a means to satisfy compatibility restrictions. Type erasure, in fact, prohibits runtime operations that depend on generic parameters, such as type tests and type casts, for example. The generic C# 2 approach changes the entire .NET Framework, including the CLR, and loses compatibility. In contrast to C# 2, NextGen 2 achieves genericity without changes to the JVM. By using a custom class loader and actually storing type information, some of the Java 5 orthogonality violations, such as subclassing of a type parameter, are no longer violations in NextGen 2. While there are some bugs in the NextGen 2 implementation, they can probably be resolved in future releases. The NextGen 2 approach proves it is possible to implement generics without modifying the virtual machine as the generic Java 5 does, but without having as many sacrifices of orthogonality as in the case of Java 5.

4.9. Limitations on Generic Type Definitions

This paper is focused on limitations of the use of Java 5 generic types in combinations with other language facilities, such as subclassing, constructor invocations, static entities, type tests, and type casts. In Java 5, a different group of limitations apply to the possibility to define generic types whatsoever. More specifically, exception classes, enumeration types, and anonymous inner classes cannot be generic in Java 5. While it is not our goal to study these limitations in detail, we provide an overview in this section.

An exception class is by definition derived from class Throwable. To handle an exception, the JVM needs to uniquely identify the particular class of each exception object, in order to match the exception object with a suitable catch clause. However, type erasure makes it impossible to distinguish between different instances of a

generic exception, because it replaces all them with the same raw type. Hence, generic exception classes are disallowed whatsoever in Java 5.

Enumeration values are implemented as static entities. As discussed in section 4.5, static access cannot be combined with type parameters. Hence, generic enumeration types are entirely disallowed in Java 5.

By definition, anonymous classes do not have names. For multiple instantiations of a generic class, the availability of a class name is essential. While it is possible to design a language mechanism that instantiates anonymous classes, such opportunity is not practically relevant. Hence, generic anonymous classes are not available in Java 5.

In contrast to Java 5, C# 2 generic types are explicitly represented at runtime. Hence, C# 2 supports generic exception classes, while Java 5 does not. Unlike Java 5, C# 2 does not support anonymous classes but offers anonymous methods, and anonymous methods can be based on type parameters. In C# 2, stand-alone enumeration types cannot be directly defined with generic parameters. However, enumeration type declarations that are nested in generic classes can actually be based on those classes' generic parameters. Hence, in C# 2 generic enumeration types can be defined indirectly - through nesting in other generic types, in contrast to Java 5, which completely excludes generic enumerations.

5. Related Work

The generic Java 5 uses type erasure to implement generic types. This approach trades orthogonality for backward compatibility. In contrast, the generic C# 2 sacrifices compatibility but honors orthogonality. NextGen 2 dismisses type erasure in a bid to prove that backward compatibility can be maintained with fewer sacrifices of orthogonality. These three approaches have been already discussed in previous sections of this paper. This section is devoted to some alternative practical approaches to the introduction of generics in Java.

A naïve generics implementation technique would be to use pure code specialization to create runtime types for generic objects. Such an approach would be very similar to the C++ template expansion. Custom class loaders supplied by the developer together with all developed class files can handle the proper loading of new generic types. This method requires no JVM byte code modifications, but leads to obvious code bloating and potential performance penalties (Myers et al., 1997). An enhancement of this approach can ensure code sharing (in the style of Java 5,

but without type erasure) by preserving some additional type information at runtime. Unfortunately, such additional runtime type information may not be easily accessed from legacy runtime environments, especially those that were not designed with future extensions in mind.

Some researchers have explored alternative methods of embedding type information in objects during compile time (Viroli and Natali, 2000). This solution leads to some code expansion and runtime penalties because reflection is employed to extract type information.

PolyJ is an extension of Java 1.0 that supports generic types. PolyJ classes can run on the JVM and Java classes can work with PolyJ's classes. The problem is that in its non-generic parts, PolyJ is limited to Java 1.0 features only, and more recent language functionality is not implemented. Thus, PolyJ is backward compatible but with an outdated version of the Java language only. In PolyJ, small wrapper classes - called trampoline classes - provide access to generic base objects; this implementation is code sharing. Primitive types such as *int* are also available for parameterization - a beneficial feature in comparison to Java and NextGen 2. A variety of structures can be statically deemed type safe, which eliminates the need for runtime checks. Hence, performance has been reported to improve by up to 17% for specific cases (Myers et al., 1997).

The PolyJ compiler is available for use; however, it does not provide adequate diagnostics. While the compiler does provide information about parsing errors, in many cases no information other than “Compilation failed” is given. Our attempt to develop and test generic cases has proved to be rather difficult.

Java ParTy is an enhanced version of the Java compiler with support for parameterized types. It is a good first step in the design and implementation of generics; as such, not all features present in Java 5 are implemented in Java ParTy. The Java ParTy approach makes use of code specialization and provides compatibility with the original JVM. Instead of inducing code bloat during compilation, Java ParTy defers type creation until runtime. This approach means that instead of extra storage space, extra memory is needed to hold each individual object type. The Java ParTy compiler creates a ParTy class, which is just a regular Java class file enhanced with type information stored in the constant pool. During load time, a preprocessor takes the ParTy classes and creates parameterized class instances, in the form of regular Java classes that can be used as types. An extended class loader loads these new classes for use within the original JVM. The preprocessor steps clearly slow down

performance, but because typecasting is not required, noteworthy runtime improvements can be achieved. Additional work, such as parameterization of primitives, is claimed to be underway (Li et al., 2000). Unfortunately, the Java ParTy compiler is not available for evaluation, and further comparisons cannot be made.

All projects discussed above testify that type erasure, although a fundamental mechanism in the generic Java 5 is not the only plausible method to implement generics or to provide backward compatibility. However, it is the most straightforward one to implement within the existing Java language and its significant legacy code. Unfortunately, type erasure mandates the exclusion from Java 5 of otherwise meaningful and useful ways to use generics. Such exclusions decrease the orthogonality and simplicity of the language.

6. Conclusions

Since the very appearance of Java in 1995, the language has been widely criticized for the lack of direct linguistic support for generic programming. The need for a generic language extension has been advocated by many members of the Java community. Various experimental generic Java extensions have been proposed. The official Java 5 has finally adopted generics by following the design and implementation of generics in the GJ language. In this article, we have investigated strengths and weaknesses of the Java 5 generics.

While class composition (through subclassing) and object composition (through object nesting) are popular and widely published, little is known about the potential of generic composition. The discussion of generic composition in the Java 5 that is offered in this article can be beneficial for Java users who would like to employ generic composition in software design.

We have compared three technical approaches to genericity, as adapted in Java 5, C# 2, and NextGen 2. Type erasure in Java allows for backward compatibility, but dismisses type information at runtime. The generic C# 2 changes the original C# language to support generics, so that generic types can be made available at runtime. There is no backward compatibility preserved. NextGen 2's enhanced class loaders can distinguish types, but this technique is not without a few bugs. It is also difficult to support both legacy Java code and NextGen 2 code concurrently in one project.

We have also analyzed how the necessity to provide full backward compatibility in the generic Java 5 has turned detrimental to orthogonality. Of course,

various linguistic restrictions of the Java 5 generics have been already analyzed in the literature. What we do differently from other authors is demonstrate in greater detail how orthogonality violations stem from particular design decisions made to satisfy compatibility constraints. More importantly, we demonstrate that such restrictions are not necessary to preserve backward compatibility by comparing the Java 5 approach to several alternative approaches that preserve type information at runtime. Finally, we analyze important tradeoffs in these alternative approaches.

Orthogonality violations in the generic Java 5 result in a complex set of rules for combining language constructs; these rules are detrimental to the simplicity and ease of use of the language. Type polymorphism does not blend easily with sub-typing, regardless of the programming language, and Java 5 is not an exception. The substantial effort of the Java 5 designers to amalgamate parameterized types and sub-typing has resulted in a number of language additions and refinements. It is unfortunate that some of these additions and refinements have further complicated the language.

Acknowledgements

This material is based in part upon work supported by the National Science Foundation under Grant Number CCF - 0243284. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

Agesen O, Freund S., Mitchell J., 1997. Adding Type Parameterization to the Java Language. Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Atlanta, Georgia, 49-65.

Allen E., Cartwright R. 2002. The Case for Runtime Types in Generic Java. Proceedings of the Inaugural Conference on the Principles and Practice of Programming, Dublin, Ireland, 19-24.

Allen E., Cartwright R., 2004. Safe Instantiation in Generic Java, Proceeding of the 3rd International Symposium on Principles and Practice of Programming in Java, Las Vegas, Nevada.

Allen E., Cartwright R., Stoler B., 2002. Efficient Implementation of Runtime Generic Types for Java. IFIP WG2.1 Working Conference on Generic Programming. http://www.cs.rice.edu/~eallen/papers/nextgen-final.pdf

Ancona D., Lagorio G., Zucca E., 2000. Jam - A Smooth Extension of Java with Mixins. ECOOP Lecture Notes in Computer Science, Cannes, France, Springer Verlag.

Bank J., Myers A., Liskov B., 1997. Parameterized Types for Java. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, 132-145.

Bracha G., 2004. Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

Bracha G., Marx S., Odersky M., Thorup K., 2001. JSR 14: Add Generic Types to the Java Programming Language. http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html

Bracha G., Odersky M., Stoutamire D., Wadler P., 1998a. GJ Specification. http://www.cis.unisa.edu.au/~pizza/gj/Documents/gj-specification.pdf

Bracha G., Odersky M., Stoutamire D., Wadler P., 1998b. Making the future safe for the past: Adding Genericity to the Java Programming Language. Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, 183-200.

Cabana B., Aladgic A., Faulkner J., 2004. Parametric Polymorphism for Java: Is There Any Hope in Sight? ACM SIGPLAN Notices, Vol. 39 (12).

Garcia R., Jarvi J., Lumsdaine A., Siek J., Willcock J., 2003. A Comparative Study of Language Support for Generic Programming. Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, California, 115-134.

Ghosh D., 2004. Generics in Java and C++ - A Comparative Model, ACM SIGPLAN, Vol. 39(5).

Gosling J., Henry M., 1996. The Java Language Environment: A White Paper. Sun Microsystems. http://java.sun.com/docs/white/langenv/index.html

Gosling J., Joy B., Steele G., Bracha G., 2005. The Java Language Specification: Third Edition. Addison-Wesley. http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf

Hejlsberg A., 2004. Generics in C#, Java, and C++, Artima Developer. http://www.artima.com/intv/generics.html

ISO, 1998. Programming Languages - C++, International Standard. ISO/IEC 14882:1998.

Kamil A., 2003. Modifications to Java since 1.0, Proposed Modifications to Java, and Titanium Compatibility with Java. http://www.cs.berkeley.edu/~kamil/titanium/doc/changes.pdf

Kennedy A., Syme D., 2001. Design and Implementation of Generics for the .NET Common Language Runtime. Proceedings of the 2001 Conference on Programming Language Design and Implementation, Snowbird, Utah.

Li S., Lu Y., Wang H., Walker M., 2000. Java ParTy: Java with Parameterized Types. Project Report 28 April 2000, University of Virginia. http://www.cs.virginia.edu/~mpw7t/cs655/ParTy/JavaParTy.pdf

Myers A., Bank J., Liskov B., 1997. Parameterized Types for Java. Proceedings of ACM Symposium on Principles of Programming Languages, pp 132-145.

Naftalin, M., P. Wadler, 2006. Java Generics and Collections. O'Reilly Media, Inc.

Odersky M., Wadler P., 1997. Pizza into Java: Translating theory into practice. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, 146-159.

Omnicore Software, 2004. Java 1.5 support in CodeGuide. http://www.omnicore.com/newlanguage.htm

Palsberg J., Schwartzbach M., 1994 Object-Oriented Type Systems. John Willey and Sons, Ltd.

Sebesta R., 2002. Concepts of Programming Languages. Addison-Wesley.

Viroli M., Natali A., 2000. Parametric Polymorphism in Java: an Approach to Translation Based on Reflective Features. Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 146-165.

Yu D., Kennedy A., Syme D., 2004. Formalization of Generics for the .NET Common Language Runtime. ACM SIGPLAN Notices 39(1): 39-51.

Appendix

This Appendix contains various classes that are implemented by means of generic composition in Java 5. Generic composition is discussed in Section 2 of this paper.

```
abstract class Ring<E> {
        E e;
        protected Ring () { }
                abstract Ring<E> add(Ring<E> other);
                abstract Ring<E> mul(Ring<E> other);
                abstract Ring<E> zero();
                E get()  { return e; }
                void set(E e) { this.e = e; }
                …
}
class RingOfInt extends Ring<Integer> {
        RingOfInt add(Ring<Integer> other) {
                e = e + other.get();
                return this;
        }
        RingOfInt mul(Ring<Integer> other) { … }
        RingOfInt zero() { e = new Integer(0); return this; }
        RingOfInt() { zero(); }
        …
}
class Array<E> {
        protected Object[] data;
        Array (int size) {data = new Object[size];}
        void set(int i, E e) { data[i] = e;    }
        E get(int i) { return (E)data[i]; }
        int size() { return data.length; }
        …
}
class ArrayOfRing<E> extends Array<Ring<E>> {
        ArrayOfRing(int size) { super(size); }
}
```

```
class ArrayOfArray<E> extends Array<Array<E>> {
        ArrayOfArray(int size1, int size2) {
                super(size1);
                for (int i = 0; i < size(); i++)        {
                        set(i, new Array<E>(size2));
                }
        }
}
class Matrix<E> extends Ring<ArrayOfArray<Ring<E>>>
{
        Matrix(int size1, int size2) {
                set(new ArrayOfArrayOfRing<E>(size1, size2));
        }
        Matrix<E> add(Ring<ArrayOfArray<Ring<E>>> other)
        {
                for (int i = 0; i < get().size(); i++){
                        for (int j = 0; j < get().get(i).size(); j++) {
                                get().get(i).set(j, get().get(i).get(j).
                                        add(other.get().get(i).get(j)));
                        }
                }
                return this;
        }
        Matrix<E> mul(Ring<ArrayOfArray<Ring<E>>> other)
        {…}
        Matrix<E> zero() { … }
        …
}
class MatrixOfInt extends Matrix<Integer> {
        MatrixOfInt(int size1, int size2) { super(size1, size2); }
        MatrixOfInt fillRandom(int limit) { … }
        …
}
abstract class MatrixOfMatrix<E> extends Matrix <Matrix<E>> {
        MatrixOfMatrix(int size1, int size2) { super(size1, size2); }
}
```

```
class RingOfBool extends Ring<Boolean> {
        RingOfBool add(Ring<Boolean> other) {
                e = (boolean)e || (boolean)other.get();
                return this;
        }
        RingOfBool mul(Ring<Boolean> other) { … }
        RingOfBool zero() {       e = new Boolean(false); return this; }
        RingOfBool () { zero(); }
        …
}
class MatrixOfBool extends Matrix<RingOfBool> {
        MatrixOfBool(int size1, int size2, Boolean b) {
                super(size1, size2);
        }
}
```

Vitae

Atanas Radenski (http://www.chapman.edu/~radenski) is a Professor of Computer Science in Chapman University, California. His research interests are in the areas of programming languages, object-orientation, computer science education, parallel and distributed computing, and evolutionary computing. He has authored about 70 publications. His research has been supported by grants from the National Science Foundation, from the National Aeronautics and Space Administration, and from other agencies.

Jeff Furlong (http://www.ics.uci.edu/~jfurlong) is a PhD student at the University of California, Irvine. His research areas have been in optimized compilers, embedded systems, FPGAs, ASICs, and hardware accelerated encryption.

Vladimir Zanev (http://csc.colstate.edu/zanev) is a Professor in Computer Science at Columbus State University, Columbus, Georgia. His research interests are in the areas of programming languages, pervasive computing, software engineering, and simulation. He is author or co-author of over 40 publications. He is a senior member of IEEE and ACM.