Coping with Java Programming Stress

Programmers who use Java know that it's a good language, but it isn't ideal. Being aware of Java's weaknesses, like its protected access and constructor confusion, will help you deal with them more intelligently.

Roger T. Alexander
George Mason University

James M. Bieman
Colorado State University

John Viega
Reliable Software Technologies

Many developers view Java as the language solution to complex software engineering problems. They expect Java programs to resist system crashes, to be written once and run everywhere, and to withstand malicious attacks. For the most part, these expectations are reasonable. Java has many attributes that promote reliable, bug-free software: memory management to prevent memory leaks, strong type checking to prevent the misuse of objects, and built-in support for exception handling. Java's virtual machine model increases portability and its security model provides a degree of safety when importing externally developed code. All these features are a great improvement over C++, Java's nominal predecessor. Indeed, initial experimental results show greater programmer productivity and fewer program bugs for development with Java versus C++.[1]

Unfortunately, however, no language is ideal, and some features of Java contribute to rather than alleviate programmer stress because they create obscure places for bugs to hide. We have identified seven features that can lead to particularly resistant bugs. Our goal is not to indict Java—we are strong supporters, and our own organizations have adopted Java as their primary programming language. Rather, we want programmers to better understand Java's weaknesses and know how to cope with them. In some cases, the strategies we suggest can prevent the weakness from affecting implementation. In other cases, they can minimize the damage. By being aware of these pitfalls and coping mechanisms, programmers can make sure that Java's design flaws don't make implementation more painful than it has to be.

ILLUSORY PROTECTION

The term "protected" implies support for encapsulation. When you see it before a program component, such as a variable or method, you naturally assume that visibility to other components is restricted. That is the purpose of encapsulation—to guard the integrity of the protected component or the entity that owns it. Once components outside that visibility border have access to the protected member, that integrity cannot be guaranteed. This is the case in Java: The visibility hole for members specified as having protected access is so large that protection is merely an illusion. A similar problem occurs for class members not specified as having a particular access (protected, public, or private).

Like C++, in Java "protected" means access to other members of the same enclosing class and to members of its descendants via inheritance. Such access increases the coupling between class definitions, but when an object references a superclass's variable, it is really just referencing part of its own state. Java supports encapsulation, but it also grants the same access to members of any class in the same package as the class having the protected member. Thus, any class with the same package designator can read and write to protected fields in any other class with the same package designator.

This creates two kinds of undesirable coupling: com-

0018-9162/00/$10.00 © 2000 IEEE

```
/* A class with a protected VIN Field */

package autos;

public class Vehicle{
   private double speed;
   private double direction;
   private String ownerName;
   protected int VIN;
   private static int highestVIN = 0;

   public Vehicle(){highestVIN++ ; VIN = highestVIN;}
   public Vehicle(String name) {this(); ownerName = name;}

   public void setSpeed(double s) {speed = s;}
   public double getSpeed() { return speed;}

   public void setDirection(double d) {direction = d;}
   public double getDirection() { return direction;}
}
```

```
/* A Rogue Class File */

package autos; /*** gains access to VIN fields by declaring itself in the
                   targeted package ***/

import autos.Vehicle;

public class RegisteredVehicle {

   static public void main(String[] args) {
     Vehicle v1 = new Vehicle("George");
     v1.setSpeed(49.5);
     v1.setDirection(45.0);
     v1.VIN = v1.VIN * 10; /**** We multiply and change a VIN ****/
   }
}
```

Figure 1. Why protection is weak with Java's protected access. The term "protected" implies support for encapsulation, but in this example, the RegisteredVehicle class breaks the encapsulation of the protected instance variable VIN in the Vehicle class. As a result, the RegisteredVehicle class can circumvent any constraints imposed in VIN by Vehicle and possibly make the state of a Vehicle instance inconsistent. [Example from* The Java Programming Language*, 2nd ed., K. Arnold and J. Gosling, Addison-Wesley, Reading, Mass., 1997]

mon coupling between all objects in the same package that reference a protected instance, and content coupling when objects reference a protected method that implements representation-dependent behavior.

The result is that any change to a protected member can ripple across to an unlimited (and possibly expanding) number of classes with the same package designator. And any component with the same package designator can modify a protected variable and force objects into invalid states.

Figure 1 shows how Java's access rules fail to support encapsulation when a new class is added. In the Vehicle class, the protected instance variable VIN represents a Vehicle instance or object's vehicle identification number. VIN should be unique for each Vehicle object and should not change during that object's life. These conditions are the Vehicle object's implied variants.

However, because the RegisteredVehicle class is in the Rogue Class File and is a member of the autos package, it can access the protected variable VIN and possibly modify it, which in turn can violate the implied invariants of the Vehicle object described earlier. This object's behavior is now quite unpredictable.

Certainly, if used with care, a package can define a collection of closely related abstractions that honor each other's semantics and consistency rules. The point is that Java cannot enforce such practices. You must rely on local honored conventions, such as coding standards, which may fail to prevent inappropriate access.

It is, of course, convenient to be able to add a new class into a package simply by using the package designator in the class code. Unfortunately, this convenience comes at the cost of encapsulation and safety. An arbitrary third party unaware of any established convention or policy could add a class just as easily. New classes added to the same package thus gain complete access to all protected members of every other class in the named package. And these new classes can subsequently violate (inadvertently or deliberately) any conventions or policies.

How to cope. Regrettably, the only way to protect a member from undesired access is to avoid using protected access. Even though you often want descendant classes to access protected members, there is just no way to restrict access to the descendants only.

```
class Super {
   Super() { printThree(); }
   void printThree() { System.out.println( “three” );
}

class Test extends Super {
   int indiana = (int)Math.PI; // That is, pi=3 in
                               // Indiana.
   public static void main( String[] args ) {
      Test t = new Test();
      t.printThree();
   }
   void printThree() { System.out.println( indiana ); }
}
Produces the following output:
   0
   3
```

Figure 2. An example of the complex order of initialization and construction that causes constructor confusion. A constructor, Super(), causes an uninitialized variable, indiana, to be used, when the program initializes a subclass, Test. [Example from* The Java Language Specification*, J. Gosling, B. Joy, and G. Steele, Addison-Wesley, Reading, Mass., 1996, p. 231.]

Until the nature of protected access in Java changes, we suggest treating protected access as if it reads “unprotected.” Make no assumptions about the integrity of any class with protected members.

CONSTRUCTOR CONFUSION

One of Java’s advertised strengths is that it initializes all variables before the program uses them. Thus, in principle, a program will invoke a class’s methods only after it has initialized all class instance variables. However, the semantics of initialization and construction in Java are not that simple. For example, a program can use instance variables before it builds the object that owns them.

The confusion results in part from the distinction between variable initialization and class construction and the order in which they can occur. When it creates a new class instance, the program first initializes variables local to that class. It then executes superclass constructors and explicitly initializes any local variables. Finally, it executes the local constructor, if it is present. A constructor can call methods, which the program can override in a descendant class. When a superclass constructor calls an overriding method while the program is building a descendant class object, the overriding method will execute before the program finishes initializing the descendant class instance. Because the construction process has not set the local variables that the overriding method can use, strange and unanticipated behavior can result.

Figure 2 demonstrates the complex order of initialization and construction and the ensuing confusion. The first statement in the method main of the Test class creates a new Test object. The program then initializes the instance variable indiana to the default value 0, deferring the explicit initialization to the value of Math.PI. The program continues by invoking the constructor of the superclass, Super(), which in turn invokes the PrintThree() method. The method invoked is not the PrintThree() method within Super, however, but the PrintThree() method in the Test class.

The program invokes the method even though it has not completely initialized indiana. Thus, PrintThree() prints a 0, which is indiana’s current value. The program then regains control from *Super*’s constructor and initializes indiana to the explicit value Math.PI (the floating-point value of π becomes the integer 3). If the Test class has a constructor, the program would run it now and complete the building of the Test object (*t*). The program then invokes printThree() of the Test object, which prints out the current value of indiana, now 3.

Methods that execute before initialization or construction are dangerous at best. Their behavior is likely to invalidate assumptions made by the authors of both the parent and descendant classes. When a base class constructor calls a method, unless the constructor invokes only *final* methods, the method defined in the base class may not be the one that actually executes. When this happens, the assumptions about the called method aren’t likely to hold.

How to cope. One approach is to require that all method calls in constructors invoke only local methods designated as *final*. This will not solve the problem, however, unless all local method calls made from a constructor result in the execution of only methods that are also defined to be final. This makes it extremely difficult to ensure correctness if you are designing a descendant class. You must have a detailed understanding of the semantics of the implementation of all ancestor classes—particularly how an overriding method affects the parent class’s state-space and which methods could possible execute in the unconstructed descendant class object.

This constructor confusion is likely to be the source of many faults, particularly if you have a C++ background, since how C++ constructors deal with local method calls is nearly the opposite of how Java constructors deal with them. For example, suppose the program is constructing an instance of a derived class in C++. A call from a base-class constructor to a polymorphic (virtual) method defined in the base class always results in the execution of the base-class method, even when the derived class has an overriding definition of the called method. This C++ construction behavior is in stark contrast to that in Java.

FINALIZATION FOLLIES

Because of Java’s mandated garbage collection, you can ignore the details of memory management.

Unfortunately, you must still manage the ownership of other resources. Thus, you must deal with many of the complex issues that C++ programmers address using destructors. Although memory leaks will not occur, scheduling the execution of Java class finalizers,[2] Java's form of destructor, can cause other resource leaks.

Java finalizer methods run when the program is through with an object and must release resources the object still holds. However, unless explicitly invoked, a finalizer runs only during garbage collection, rather than when the object loses its last reference. Thus, finalizers run at unpredictable times, just like garbage collection. The uncertainty about the time that the finalizer runs can lead to trouble. Suppose a class has a constructor that allocates a network connection and a finalizer that closes down the connection. Many systems map each network connection to a file pointer in the operating system. Generally, relatively few file pointers can be open at once. If a program instantiates and then discards a large number of these objects before the garbage collector calls any finalizers, any attempt to create a new file or network connection will fail.

How to cope. Don't count on finalizers executing in a timely manner. In fact, there is no guarantee that finalizers will ever run at all. For example, when the program exits, no finalizers will run for any objects that have become garbage since the last collection, unless the programmer explicitly ensured that the program called System.runFinalizersOnExit(true). Even that is no guarantee that the finalizer will run. For example, the current version of Sun's Java virtual machine will not run the finalizer if an outside signal terminates it.

Also, don't expect finalizers to execute in a deterministic order. For example, finalizers will not necessarily run in the order that the objects became garbage; the actual order is unspecified.[2]

The best strategy is to avoid finalization if possible. If you must use it, and your finalizers must be called in a timely manner, explicitly call the garbage collector that will invoke the finalizers. For this to work, you must know beforehand that a given object will be available for finalization, which means that you must track all references to that object. An explicit call to the garbage collector will not invoke an object's finalizer if any references to the object remain.

An alternative is to add public methods that the program can call to release resources an object is holding even though it no longer needs them. However, again, you must track all references to the object holding the resources and assign the responsibility for calling the methods. This isn't trivial, and an error can be costly: A resource could be deleted when clients are still using it.

INHERITANCE WITHOUT SPECIALIZATION

Subclasses are descendants of other defined classes. Java and other object-oriented languages let you substitute a subclass object for a superclass object. However, you must satisfy certain properties to guarantee that your substitution is safe.[3] One safe substitution is when the subclass is a specialization of the superclass. For example, a Cartesian point with color attributes can be a specialization of a Cartesian point without color. You can then substitute a colored point for a plain point because any behavior of plain points also applies to colored points.

Improper use of subclasses in Java can be an especially troublesome source of bugs that are difficult to diagnose and correct.

Problems can occur when a subclass is not a true specialization of its superclass. Consider the java.util.Stack class, which is part of the java.util package. Class java.util.Stack is a subclass of java.util.Vector. Stack defines common stack methods such as push(), pop(), and peek(). However, because Stack is a subclass of Vector, it inherits all the methods Vector defines. Thus, you can supply a Stack object wherever the program specifies a Vector object. A program can insert or delete elements at specified locations in a Stack object using Vector's insertElementAt or removeElementAt methods. It can even use Vector's removeElement method to remove a specified element from a Stack object without regard to the element's position in the stack.

Consequently, the java.util.Stack can exhibit behavior that is not consistent with the notion of a stack as a last-in, first-out entity. In addition, a program can access all the Vector operations on Stack objects directly when the Stack objects are not being substituted for Vector operations.

A stack is not a specialized vector, and it should not inherit vector operations. Instead, a vector should be a hidden, private representation of a stack. Stack objects cannot then export inappropriate vector operations. This preferred design uses aggregation, which lets you use inheritance and polymorphism to replace the vector representation with alternative implementations. If you use inheritance properly, the design will be more flexible and efficient.

How to cope. In general, substitution will be safe if you use a subclass when the derived class is a specialization of the superclass. In this "is-a" relationship, subclass objects behave similarly to superclass objects but have additional features, operations, or both. If you are unsure how to use inheritance, Bertrand Meyer offers a good taxonomy that classifies both proper and improper uses.[4] Improper use of subclasses in Java can be an especially troublesome source of bugs that are difficult to diagnose and correct. Java does not provide the mechanisms that C++ does to make improper subclasses a bit safer. In particular,

```
import java.util.List;
import java.util.LinkedList;

/*
 *From the Java Collections Framework:
interface List {
  public void add( Object element );
  public Object get( int index );
    ...
  }
*/

class StringListExample{
  public static void main(String[] args){
    List l = new LinkedList();
    for(int i=0;i<args.length;i++){
      l.add(args[i]);
      System.out.println((String)l.get(i));
} } }
```

(a)

```
class StringList{
  private List my_list;
  public StringList() {my_list = new LinkedList();}
  public void add(String elem){my_list.add(elem);}
  public String get(int index) {return (String)my_list.get(index);}
  ...
}
```

(b)

```
class RunTimeList{
  private List my_list;
  private Class thisClass;
  RunTimeList() {my_list = new LinkedList();}
  void add(Object elem) throws BadElement{
    if (my_list.isEmpty()) thisClass = elem.getClass();
    if (thisClass != elem.getClass())
      throw new BadElement(thisClass, elem.getClass());
    my_list.add(elem);
  }
  Object get(int index){
    return my_list.get(index); // Java won't let us cast back
                               // to "thisClass" here.
  }
}
```

(c)

Figure 3. Three possible strategies for creating an object from the Java Collections Framework class LinkedList, which is meant to be a homogeneous list of String elements. (a) Use a universal list, a list of Object elements, to hold Strings. (b) Create a special-purpose StringList that can hold only String objects. (c) Use a list that sets the class of its contents when the program inserts the first element at runtime. None of these approaches is ideal.

there is no mechanism to hide inherited members or to break the type relationship with the subclass's parent. Thus, there is no way to prevent a client from seeing a descendant as an instance of its base class.

One approach is to provide overriding methods for each inherited method and implement them by throwing invalid method exceptions. Unfortunately, you cannot override any methods that are declared as final in a parent class. You must also declare the exception in the parent class unless you throw an unchecked exception, such as those derived from RuntimeException. Another possible solution is to use some sort of assertion mechanism to restrict the use of inappropriate inherited methods.

CONTAINER LIMITATIONS

Java provides little flexibility for creating specialized, homogeneous container classes. You must either use containers that can hold anything or write special-purpose classes that define containers for each kind of element. Java does not yet support type-safe parameterized classes, as C++ does with templates or Ada does with generics. Instead, it provides the universal base class Object, a superclass to every class.

Suppose you want to create an object from the Java Collections Framework class LinkedList, which is meant to be a homogeneous list of String elements. In C++, you can simply instantiate an object of type LinkedList<String>, and the compiler will ensure that

only String objects are inserted into the LinkedList<String> object. In Java, on the other hand, you must resort to one or a combination of the options in Figure 3:

- Instantiate a LinkedList object that accepts any object whatsoever (objects of class Object) and place only objects of class String into it (Figure 3a).
- Write a special-purpose adapter class with the functionality of a LinkedList class that operates only on String objects (Figure 3b).
- Write a LinkedList class that, at runtime, sets the type of inserted objects according to the class of the first object inserted (Figure 3c).

How to cope. Unfortunately, there is no workaround. Our best advice is to weigh the risks of each approach and then proceed with caution. The first approach is the most common, but provides no type safety. You must ensure that the program inserts only objects of the desired class into a list and explicitly puts objects from the list back into the desired class. Because the program must return the objects to the desired class—perform casting—at runtime, a method might insert non-String objects into the list. To prevent these kinds of errors, you must track type information.

The second approach ensures that the program can type-check calls to container operations properly at compile time. In addition, the special-purpose class can perform all casting. Of the three approaches, it offers the most type safety, but it does so at the expense of proliferating nearly identical classes. To minimize code replication, you can have the adapter provide the necessary interface, but implement it in terms of LinkedList. This approach imposes the required level of type safety while reusing the existing available implementation.

The advantage of the third approach is that you can implement it using only one class—the class that captures the type of the first object to be inserted into the list. However, again, you must explicitly cast elements retrieved from the list back into the desired class, so the third approach has similar drawbacks to the first.

NOT-SO-FINAL PARAMETERS

In Java, a method can change the state of any object of the class that the method is a member of. Thus, to be safe, a client must assume that a method invocation on an object can modify that object's state. If you are programming that client, you must look at the implementation of the called method to really know if the object's state has changed. Unfortunately, because you generally won't have access to the implementation, you must trust the documentation, which, of course, imposes no guarantees or constraints on the called method's implementation.

Your only solution is to document all side effects and ensure that method implementations remain consistent with the documentation. Even this will not work on Java components obtained externally.

Java 1.1 lets you declare a method's formal arguments to be final, ensuring that the state of the argument cannot change. Regrettably, the guarantee applies only to the state of the parameter variable itself, not to the state of any class it references. Java does not let you change the value of a formal argument that is a reference to an object even though that argument is declared to be final, but it does let you change the state of the object being referenced. You can use the final parameter variable to invoke any method defined in the object's class. Thus, when a program supplies an object reference as an argument to a method call, the state of the referenced object argument can change, even if the associated formal parameter is designated as final. You are forced to trust the called method, inspect the method if possible, or add code to verify that no state changes have occurred and add error-handling code.

The protection the final parameter designator provides has severe limitations that could cause a system to enter an inconsistent state.

The protection the final parameter designator provides has severe limitations that could cause a system to enter an inconsistent state. Testing alone cannot guarantee that all methods behave as expected.

How to cope. Your only protection is to make sure that methods contain correct documentation that explicitly describes the effects of a particular method call. These effects include both those on a given instance and those on any instances passed via object references as actual arguments.

INITIALIZATION DIFFUSION

The JDK 1.1 Java Language Specification includes code blocks that initialize the state of object instances, which are similar to blocks that initialize class state. You can write an instance initialization block simply as an unlabeled block of code that appears at any location in a class definition. There can be multiple instance initialization blocks, possibly distributed at various locations within a class. The program executes initialization blocks in the order they appear in a class.

Figure 4 demonstrates how initialization blocks diffuse initialization across a class. When the program creates a new Vehicle (or VehicleDiffusion) object, we want to increment the static variable highestVIN so that we can use a unique VIN for each new Vehicle object. Class Vehicle updates highestVIN in the expected place—within the constructor method Vehicle(). Class VehicleDiffusion updates highestVIN in an initialization block rather than in a constructor. To further confuse code readers, the program places the update in a location apart from the constructors.

```
/* "Normal" initialization in class Vehicle */
class Vehicle{
   private double speed;
   private double direction;
   private String ownerName;
   private int VIN;
   private static int highestVIN = 0;

   public Vehicle(){highestVIN++; /***increment highestVIN ***/
                    VIN = highestVIN;}
   public Vehicle(String name) {this(); ownerName = name;}

   public void setSpeed(double s) {speed = s;}
   public double getSpeed() { return speed;}

   public void setDirection(double d) {direction = d;}
   public double getDirection() { return direction;}
}
-------------------------------------------------------------------------------------------------------------------

/* Diffused initialization in class VehicleDiffusion */
class VehicleDiffusion{
   private double speed;
   private double direction;
   private String ownerName;
   private int VIN;
   private static int highestVIN = 0;

   public VehicleDiffusion() {VIN = highestVIN;}
   public VehicleDiffusion(String name) {this(); ownerName = name;}

   public void setSpeed(double s) {speed = s;}
   public double getSpeed() { return speed;}

   {highestVIN++;} /*** increment command moved here ***/

   public void setDirection(double d) {direction = d;}
   public double getDirection() { return direction;}
}
```

Figure 4. Initialization diffusion caused by instance initialization blocks. Vehicle and VehicleDiffusion objects exhibit identical behavior although VehicleDiffusion moves the command highestVIN++ in the Vehicle constructor to an initialization block. [Example from* The Java Programming Language*, 2nd ed., K. Arnold and J. Gosling, Addison-Wesley, Reading, Mass., 1997.]

The semantics of creating a new Vehicle or VehicleDiffusion object are essentially identical.

Instance initialization blocks, introduced with Java 1.1 syntax and semantics,[5] add two new sources of program errors. First, initialization code is distributed between constructors and initialization blocks, which can be distributed throughout a class. Thus, to understand the full instance initialization and construction process, you must understand the semantics of constructors and instance initialization blocks. This means scanning an entire class definition looking for instance initializers, analyzing the semantics of each initializer and its order of execution, and then analyzing the class construction methods' semantics. This process is tedious and error-prone when you have many instance initializer blocks.

Second, the syntax of instance initializer blocks can lead to errors. The only syntactic difference between an instance initializer block and a static initializer is the keyword "static." If static appears immediately before a class-level block, the block defines a static initializer.

If the static keyword is missing and no other lexical element appears in its place (such as a method signature), the block defines an instance initializer. In addition, an instance initializer's structure is identical to a method definition that is missing its body. Unfortunately, it is easy to accidentally delete a single word like "static" or a line that defines a method interface. These simple editing errors turn a static initializer or a method into an instance initializer, which might compile without warning. Debugging these errors is difficult.

How to cope. Avoid using instance initializer blocks and put initialization code in constructor bodies. If you *must* use an initializer block, use only one per class and locate it close to the class's constructors.

OTHER WORRIES

In addition to these seven design weaknesses, Java has a number of syntax quirks and inherits many of the syntax problems of C++ and C.[6] Other sources of programming stress include:

No separate class-specification. Java does not allow a class specification to be separate from its implementation because both the public class interface and method bodies must be in one file. Thus, anyone accessing the public interface can view its implementation. Knowledge of the implementation makes it possible to

write software components that depend on this implementation, which violates encapsulation principles.

Making code easy to understand is key to maintenance and reuse. Programmers writing a client must now wade through an entire class body just to view the public interface of a potential server class. The solution to this problem would not require separate interface and implementation files for each class, as the .h and .cpp files in C++. Rather, Java could have a separate syntactic mechanism within a class that specifies its interface. The corresponding implementation might still appear in line within the class body.

The Javadoc facility can extract interface information. However, Javadoc relies on a class's author to write appropriate Javadoc comments and keep them current. Comments and code usually diverge over time, however, which means that the Javadoc interface information will eventually be inaccurate.

No support for assertions. Object-oriented languages, such as Eiffel and Clu, support assertions.[7,8] Properly used, an assertion mechanism can increase a component's quality and correctness. Programmers can specify pre- and postconditions for methods and data invariants for class state variables.

Java does not have a built-in assertion mechanism, although tools are freely available for assertion support, such as Reliable Software Technologies' AssertMate[9] and Reliable Systems' iContract.[10] The Java FAQ Web site[2] tells you how to construct a rudimentary assertion mechanism. Eventually, we hope to see Java support its own assertion mechanism.

Array type-checking failures. Static type-checking limitations can allow programs with obscure array type errors to compile. Although dynamically binding subclass objects is usually type-safe and flexible, it can lead to a type clash or covariance problem if subclass objects are bound with arrays. Generally, you can supply an instance that is a subclass of the declared type of the formal parameter as an argument in a method call. However, sometimes dynamic binding conventions and associated type-checking rules cannot detect type errors.

Figure 5 shows why. In this program, all actual parameters are instances of the formal parameters' subclasses. The program compiles even though it contains a serious type error. No type errors are detected during compilation because argument anArrayOfB is an instance of a subclass of formal parameter *x*'s class, A[], and the type of argument *a* exactly matches the type of formal parameter *y*. Yet, the program fails at runtime.

```
class A { . . . }

class B extends A { . . . }

class ArrayConfusion {
   static void proc(A[] x, A y) {
     x[0] = y;
   }
   public static void main(String args[]) {
     B[] anArrayOfB = new B[5];
     A a = new A(5);
     proc(anArrayOfB, a);
   }
}
```

Figure 5. A sample array type-checking problem. ArrayConfusion.main calls ArrayConfusion.proc with two arguments: anArrayOfB and a, an instance of class A. ArrayConfusion.proc has two formal parameters, x, an array of A objects, and x, an A object. Java's static type-checking does not catch the type error when the first argument to ArrayConfusion.proc is an array of class B and the second argument is an object of class A. The assignment x[0] = y raises a runtime exception. The assignment x[0] = y raises a java.lang.ArrayStoreException because an array of class B objects cannot store an object of class A. A is not a subclass of B. This program has an illegal assignment of a supertype to a subtype variable and fails at runtime, yet it compiled with no detected errors.

The shortcomings we have identified are worrisome because Java is intended for the development of concurrent, distributed, and critical systems. Our coping suggestions can remedy or soften the effects of these problems or help you avoid them, but changes to the language itself would offer a more effective long-term solution. Here's our wish list:

- Replace the package-level component of protected access with a mechanism that lets a class specify what other specific classes, or group of classes with a particular characteristic, can access it.
- Make default access private.
- Change the semantics so that constructors cannot invoke any methods, directly or indirectly, in subclasses.
- Support templates or generics.
- Make the final parameter designator ensure that a method with a final parameter cannot modify the state of objects the parameter references. Add

A Coping Checklist

- ✔ Avoid using protected or package-level (default) access; declare all members as either private or public.
- ✔ Take extra care in understanding the construction of new objects that override superclass methods and instance variables.
- ✔ Use container classes with caution. Java type checking is not effective here.
- ✔ Use the subclassing mechanism only to define specializations of a superclass.
- ✔ Explicitly force finalizers to run when you want them to.
- ✔ Use inheritance only to model is-a relationships.
- ✔ Document all side effects, and make sure that the documentation is consistent with the code.
- ✔ Avoid using instance initializer blocks.

a constant method designator to ensure that the state of the object containing the method does not change.
- Require a keyword to specify an instance initializer and allow only one such initializer block per class.

It would be unrealistic to expect future versions of Java to incorporate all these changes. A language many people use must satisfy myriad, often competing, interests. Some of our changes could be made with little effort, but others will require major effort and probably some research.

An immediate first step would be to use testing and static program analysis to identify programs with the problems we've described. For example, static analysis can easily identify programs that use instance initializer blocks, protected and package (default) access, or both. Another immediate goal is to develop tools and techniques, through either static analysis or testing, that can identify programs suffering from the Java weaknesses we have identified. ❖

Acknowledgments

We thank Reliable Software Technologies, Sterling, Va., and the University of Maryland at College Park for their generous support during Jim Bieman's sabbatical when we prepared this article. We also thank Jefi Ofiutt and Gary McGraw, whose comments on earlier drafts greatly improved the presentation. Finally, we thank the anonymous *Computer* reviewers for their comments, which greatly improved both content and presentation.

References

1. G. Phipps, "Comparing Observed Bug and Productivity Rates for Java and C++," *Software: Practice and Experience*, Apr. 1999, pp. 345-358.
2. P. van der Linden, "Frequently Asked Questions (with Answers) for Programmers Using the Java Language," http://www.afu.com/ javafaq.html.
3. G. Leavens, "Modular Specification and Verification of Object-Oriented Programs," *IEEE Software*, Nov./Dec., 1991, pp. 72-80.
4. B. Meyer, "The Many Faces of Inheritance: A Taxonomy of Taxonomy," *Computer*, May 1996, pp. 105-108.
5. K. Arnold and J. Gosling, *The Java Programming Language*, 2nd ed., Addison-Wesley, Reading, Mass., 1997.
6. H. Thimbleby, "A Critique of Java," *Software: Practice and Experience*, May 1999, pp. 457-478.
7. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, MIT Press, Cambridge, Mass., 1986.
8. B. Meyer, *Eiffel: The Language*, Prentice Hall, Upper Saddle River, N.J., 1992.
9. J. Payne, M. Schatz, and M. Schmid, "Implementing Assertions for Java," *Dr. Dobb's J.*, Jan. 1998, http://www.ddj.com/articles/1998/9801/9801d/9801d.htm#re1
10. R. Kramer, "iContract—The Java Design by Contract Tool," *Proc.Technology of Object-Oriented Languages and Systems, (TOOLS-28, 98)*, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 295-307.

Roger T. Alexander is a member of the research faculty at George Mason University and a principal member of the technical staff at the Software Productivity Consortium. His research interests include object-oriented software testing, architecture testing, specification-based testing, software reliability, and software metrics. He received an MS in software engineering from George Mason University, where he is a PhD candidate in computer science. Alexander is a senior member of the IEEE. Contact him at rtalexander@computer.org or ralexand@gmu.edu.

James M. Bieman is an associate professor of computer science at Colorado State University. His current research interests are evaluating object-oriented designs and developing ways to quantify design attributes in terms of architectural structures and patterns. He is also studying the relationship between design attributes and external quality attributes such as maintainability, testability, and reliability. Bieman received a PhD in computer science from the University of Southwestern Louisiana. He is a senior member of the IEEE and is chair of the Steering Committee for the IEEE-CS International Symposium on Software Metrics. Contact him at bieman@cs.colostate.edu.

John Viega is a senior research associate and consultant at Reliable Software Technologies. He is the principal investigator on a DARPA-sponsored grant for developing security extensions for standard programming languages. He also writes a bimonthly column on software security assurance for developers (http://www.ibm.com/developer/security). Viega received an MS in computer science from the University of Virginia. Contact him at jviega@rst.com.