Cover Feature

Behavioral Specification of Distributed Software Component Interfaces

Design by contract solves the problem of behavioral specification for classes in an object-oriented program, which will ultimately lead to more robust and maintainable distributed software components.

Cynthia Della Torre Cicalese
Global InfoTek Inc.

Shmuel Rotenstreich
The George Washington University

With the popularity of the Web and the Internet, networked computers are finding their way into a broader range of environments, from corporate offices to schools, homes, and shirt pockets. This new computing model fosters the development of *distributed software components* that communicate with one another across the underlying networked infrastructure.

A distributed software component can be plugged into distributed applications that may not have existed when it was created. The intention is that many developers will reuse distributed software components to build new systems. For a developer to know what to expect from a distributed software component, the design must clearly document both the component's interface (operation signature and calling conventions) and the operations' behavior.

An *interface definition language* usually is used to describe a distributed software component's interface. However, a notable limitation of current IDLs is that they generally only describe the names and type signatures of the component's attributes and operations. Current IDLs don't formally specify the behavior of the software component's operations.

Inadequate specification of reusable software can result in disaster. The failed launch of the $500 million Ariane 5 in 1996 is a case in point.[1] Code originally intended to convert a number less than 2^{16} from a 64-bit floating-point number to a 16-bit unsigned integer was applied to a greater number, causing the software, and the rocket, to crash.

Design by contract[2] solves the problem of behavioral specification for classes in an object-oriented program. In this article, we discuss how to apply design by contract to distributed software component interfaces.

Biscotti (behavioral specification of distributed software component interfaces) is an extension of Java that enhances Java remote method invocation interfaces with Eiffel-style preconditions, postconditions, and invariants. Integrating this specification information into the language complements other Java features, such as reflection, which is fundamental to JavaBeans software components. Java developers easily transition to using Biscotti because it is an extension of a familiar programming model.

DISTRIBUTED SOFTWARE COMPONENTS

Ideally, a distributed software component has the following characteristics:

- a well-defined, well-documented interface defining the services it provides,
- software independence from its clients,
- remote access,
- encapsulated implementation,
- robustness, and
- reusability.

The object-oriented models that are the basis of most common distributed software component technologies foster reuse through the abstraction and encapsulation characteristics of objects. A key strength of the object model is its ability to deal with distributed computing's inherent complexity.

Remote-calling communication model

The remote-calling model is typically used to interact with distributed software components. Figure 1 shows the interactions in this model:

0018-9162/99/$10.00 © 1999 IEEE

1. A program is executing and encounters a call to an operation in a remote software component.
2. The operation parameters are marshaled (put in a format suitable for transmitting them across the network) and sent to the server responsible for executing the remote software component.
3. The remote server unmarshals the parameters (puts them in a format suitable for computations on the server machine) and passes them to the remote operation.
4. When the remote operation is finished executing, its return value and any parameters that changed value are marshaled for return to the client.
5. The client unmarshals the return value and returned parameters and continues execution.

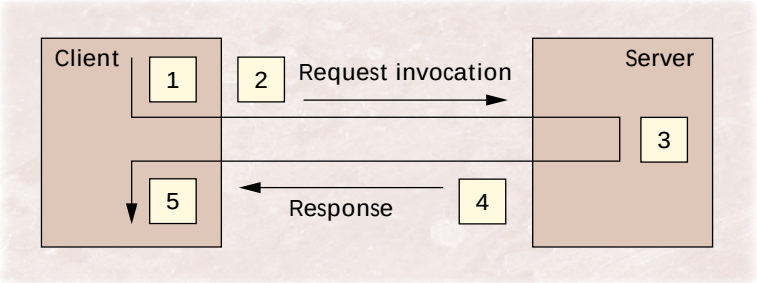

Figure 1. Remote-calling communication model.

Generally, a tool-generated *client stub* code performs the client-side marshaling and unmarshaling, and a tool-generated *server skeleton* code performs the server-side marshaling and unmarshaling. The client stub and server skeleton interface between the application layer and distributed communication layer.

Interface definition languages

Two factors are required for a system's software components to work together. First, the components must have a common understanding of how to communicate with one another. The network infrastructure provides this service. Second, each component that provides a service must have a well-defined interface indicating how to access the service.

Most popular distributed-computing frameworks satisfy this second requirement by providing an IDL that documents a software component's services in a location-, platform-, and programming-language-independent fashion.

For a given IDL, compilers translate interface definitions into client stub and server skeleton code in a target programming language. This code provides the marshaling and unmarshaling glue between the infrastructure and the application developers' client and server code. Some IDLs are used with a specific programming language, but others can be used with various programming languages. For these IDLs, multiple programming language bindings specify the mapping from the IDL to the target programming language client stubs and server skeletons.

Following the generalized scenario shown in Figure 2, the server developers define the server's IDL interface as a part of the system design. They use an IDL compiler for the selected server programming language to compile the IDL specification into a server skeleton. The other server code is then implemented. When deployed, the server will be accompanied by its interface IDL description. Similarly, a client developer chooses a client programming language, uses an IDL compiler to compile the IDL specification into a client stub, and the other client code is implemented.

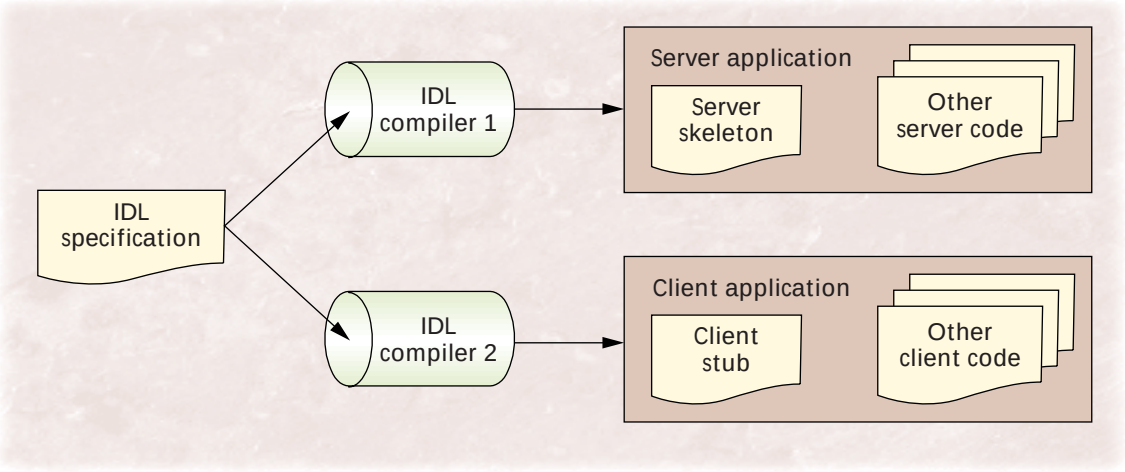

Figure 2. Server and client developers compile the same IDL specification to generate a server skeleton or client stub. These developers then write other server or client application code, which communicates with the skeleton or stub based on the IDL interface specifications.

The target client and server programming languages may be completely different, but the client can communicate with the server because they are compiled from the same IDL specification. The client developer usually knows the details of the server's interface, and writes client code expressly for communicating through that interface. However, most distributed software component technologies also provide support, at runtime, for dynamically determining the server interface's structure, as defined in an IDL, and formulating calls to server operations.

Limitations of current IDLs

Current IDLs generally describe only the names and type signatures of the component's attributes and operations. It is possible to define an interface to which two implementations with different behaviors could correspond. Naming conventions alone can't disambiguate the interface because different people can interpret the names differently. An operation's signature doesn't guarantee how the parameters will be used. The client doesn't know whether the parameters must be within certain ranges, how the operation will change the system's state, or what effect this will have on the returned values. Thus, a client could communicate with a software component through its interface and still not interoperate with it.

Distributed software component developers document behavioral information with intelligent naming conventions and abundant comments. While these are important, formal and unambiguous documentation of the operational behavior of reusable distributed software components is imperative. This is especially true if the component is reused long after it was created or if it is used by different developers than those who created it.

Thus, in addition to an interface definition, a behavioral specification is necessary. The interface's opera-

```
public class StackFullException
   extends Exception
{
}

public class StackEmptyException
   extends Exception
{
}

public interface StackInterface
   extends java.rmi.Remote
{
   boolean isEmpty()
      throws java.rmi.RemoteException;

   boolean isFull()
      throws java.rmi.RemoteException;

   void push(Object obj)
      throws java.rmi.RemoteException,
      StackFullException;

   Object pop()
      throws java.rmi.RemoteException,
      StackEmptyException;
}
```

(a)

```
exception StackFullException
{
};

exception StackEmptyException
{
};

interface StackInterface
{
   boolean isEmpty();

   boolean isFull();

   void push(in short i)
      raises (StackFullException);

   short pop()
      raises (StackEmptyException);
};
```

(b)

```
import "unknwn.idl";

[ uuid(c38cea50-498a-11d2-8043-00104b235515) ]
interface StackInterface :IUnknown
{
   HRESULT isEmpty([out] boolean *empty);

   HRESULT isFull([out] boolean *full);

   HRESULT push([in] int i);

   HRESULT pop([out] int *i);
};

[ uuid(ddd99e70-498a-11d2-8043-00104b235515) ]
library StackLib
{
   importlib("stdole32.tlb");

   [ uuid(e9868930-498a-11d2-8043-00104b235515) ]
   coclass Stack
   {
      interface StackInterface;
   };
};
```

(c)

Figure 3. Stacks defined in (a) an RMI-enabled Java interface, (b) the CORBA IDL, and (c) DCOM IDL.

tional behavior must be formally specified to convey as much information as possible without putting an unrealistic burden on the interface developer.

Distributed software component IDLs

Several distributed software component technologies have gained acceptance, including

- Sun's Java remote method invocation (RMI),
- Object Management Group's Common Object Request Broker Architecture (CORBA), and
- Microsoft's Distributed Component Object Model (DCOM).

RMI. Java's ease of use, portability, and facilities for Web-based programming make it ideal for Internet programming. Java programs are compiled into bytecode that is executed on a Java virtual machine. Since JVMs are available for most modern computing platforms and are built into most popular Web browsers, Java programs, including compiled class files, are extremely portable.

Java distinguishes between interfaces and classes. A Java interface is equivalent to an abstract class containing only abstract methods. As with IDL interfaces, a Java interface is defined independently from its implementation, a division most common OO programming languages don't make. A Java class can "implement" zero or more interfaces, but it can only "extend" (inherit from) a single superclass.

RMI supports distributed communication between objects written in Java. Java's interface construct describes distributed objects. This serves the same function as an IDL in other distributed software component approaches. The RMI interface compiler generates client stub and server skeleton bytecode. Figure 3a presents an example of an RMI interface.

The extremely flexible RMI programming environment has several powerful features. For example, it passes full objects as RMI operation parameters and return values. It serializes the object's state and passes it on to the remote object; if necessary, it also passes the object's class bytecode. Programming-language-independent IDLs limit the data types used in interfaces to the least common denominator of all supported programming languages. Because RMI uses any valid Java class for parameter or return value types, it builds more powerful interface definitions. But this power comes at the cost of interacting transparently with distributed software components implemented in other programming languages.

Java has built-in facilities for *reflection*: dynamically discovering an object's structure and using information extracted from its .class file to manipulate the object at runtime. We can write extremely flexible software with the technique. JavaBeans relies heavily upon reflection, which can be used on any Java object, including distributed RMI objects.

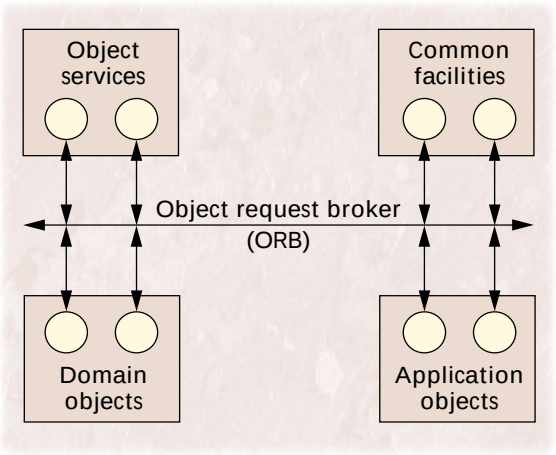

Figure 4. Object management architecture.

CORBA. CORBA is a programming-language- and platform-independent architecture for interoperable distributed software components. As shown in Figure 4, CORBA's underlying object management architecture presents the distributed-computing enterprise as distributed objects communicating over the object request broker bus. These objects, all defined in OMG IDL, fall into four categories:

- *object services* provide the basic services necessary in virtually any enterprise;
- *common facilities*, also referred to as horizontal facilities, are useful across application domains, but they are not as fundamental as object services;
- *domain objects* are useful in certain vertical areas such as finance or health care; and
- *application objects* are created for a particular end-user function.

OMG standardizes the interfaces to object services, common facilities, and domain objects, but application object interfaces are not standardized. Application objects use the object services, common facilities, and domain objects. Figure 3b presents an example of an OMG IDL interface.

Distributed objects defined using OMG IDL can be implemented in any programming language that has an IDL mapping. CORBA's specification currently includes IDL mappings to C, C++, Smalltalk, Cobol, Ada, and Java. Other language mappings, such as to Eiffel, are in development.

CORBA supports both a static invocation interface, using client stubs and server skeletons, as well as dynamic invocation and skeleton interfaces. The dynamic invocation interface uses an interface repository to allow information about distributed objects to be discovered at runtime. This information is used to

Tools can use behavioral specifications to verify that the software meets its requirements.

build dynamic operation invocations. With the dynamic skeleton interface, servers react dynamically to operation invocations, instead of using static server skeletons.

DCOM. DCOM is a distributed version of Microsoft's COM architecture that defines a binary standard for software components. As such, a COM component can be written in any programming language that compiles into a Win32 dynamic link library.

Because COM is a binary standard, originally no IDL was provided to document interfaces at the source code level. The tools had to determine how to map from a particular programming language directly into the binary format. Later, an IDL was introduced to make reusing COM components easier.

COM was extended to DCOM for distributed computing. DCOM's underlying remote procedure call mechanism and IDL are extensions of the Open Group's Distributed Computing Environment. Figure 3c shows a DCOM IDL description of a stack.

Clients use type libraries to determine the COM component's interface structure at runtime. A type library is created by compiling the component's IDL.

BEHAVIORAL SPECIFICATION

A *behavioral specification* is the formal description of what is supposed to happen when software executes. Tools can use the specification to verify, statically or at runtime, that the software meets it requirements.

Current software specification languages include Z, Larch, OBJ, and Vienna Development Method. Languages used to specify communication protocols such as the Language of Temporal Ordering Specifications, Estelle, Process Meta Language, and the Specification and Description Language are also of interest.

Eiffel's design incorporates behavioral specification constructs including instance invariants and operation preconditions and postconditions as well as general assertions, loop variants, and loop invariants. Assertions have been added to several other programming languages through comments understood by special preprocessors or by additional libraries. Observers have several ideas about why behavioral specification and the larger discipline of formal methods are still not widely used in practice.[3,4]

- The tools are immature, full of bugs, and difficult to use. They are not cost-effective, and they are not integrated with accepted tools and approaches.
- The tools and methods developed by researchers are not marketed effectively to industry.
- There are few real-world examples and success stories.
- Much of the target audience in industry does not have a sufficient mathematical background to be comfortable with using the mathematical notations.

These impediments are gradually changing, except for the concerns about the mathematical background of the target audience. If developers are unlikely to have adequate training in mathematics, those advocating formal methods must seek a compromise to ensure the acceptance of formal methods by industry.

Design by contract

Design by contract makes behavioral specification more accessible to programmers.[2] This model views the relationship between a class and its clients as a contract that takes the form of assertions: Boolean invariants, preconditions, and postconditions. Boolean invariants and preconditions document the contractual obligations a client must abide by before calling an operation in a class. When the client fulfills its obligations, Boolean invariants and postconditions document the class supplier's contractual obligations for how the operation must behave and what it must return.

For example, we could define a stack class's pop operation with the precondition that the stack must not be empty and a postcondition that the top element of the stack will be removed and returned to the client. The client's obligation is to ensure that it does not call the pop operation when the stack is empty. If that condition is satisfied, the operation supplier's obligation is removing the top element of the stack and returning it to the client.

Design by contract assumes that the programming language supports the assertions and that they appear in the source code itself. This ensures that the software and its specification remain synchronized as development and software maintenance progress. The contract plays an essential role in documenting a class. In contrast to a separate specification language, a high-level language with embedded specification constructs is more likely to be used. Developers can learn about using formal specifications just by using the high-level language, which helps bridge the experience gap. Also, integrating built-in specification constructs with other programming language features—for example, a language that supports reflection—allows the developer to write software that can discover assertions at runtime.

During testing, the system can monitor assertions embedded in the source code. When an assertion fails to hold, the program throws a runtime exception. Assertion failures help localize the sources of bugs. For example, without preconditions, a method called with unacceptable parameter values can generate a nonsensical return value, rather than dying. The program dies much later when it uses the return value. With a method precondition, however, the program detects the incorrect parameter values when it calls a

method. It throws a runtime exception at the point where the error occurs.

BISCOTTI

We created Biscotti, a Java extension, to evaluate the use of design by contract to add behavioral specification constructs to Java RMI interfaces. Figure 5 shows an example of this application.

Design goals

The design goals of this work are to

- enable behavioral specification of Java RMI interfaces;
- make behavioral-specification constructs a natural extension to the Java language that the interface developer is likely to accept and use;
- write the behavioral specification in terms of the operations accessible through the interface so the developer doesn't need to build a separate interface specification model;
- carry behavioral-specification constructs through to the Java code, allowing behavioral specification of a class that implements an interface;
- use Java's exception-handling capabilities to enable runtime monitoring of specification violations;
- ensure that a valid Java program is also a valid Biscotti program;
- ensure that the JVM doesn't need any changes to run a Biscotti program; and
- use Java's reflection facilities to make preconditions, postconditions, and invariants visible at runtime.

Grammar

Biscotti's grammar is identical to Java's with three exceptions:

- It introduces six new keywords: *invariant*, *requires*, *ensures*, *then*, *old*, and *result*;
- Biscotti method declarations can specify optional precondition and postcondition declarations; and
- Biscotti interfaces and classes can specify an optional class and instance invariant.

Preconditions are introduced with the keywords *requires* or *requires else*, and postconditions are introduced with the keywords *ensures* or *ensures then*; the longer form is used to indicate that the operation is overriding an inherited operation. Invariants are introduced with the keywords *invariant*, for instance invariants, or *static invariant*, for class invariants. The *old* and *result* keywords are used in postconditions to refer to the value of a parameter when the method was entered and the return value of the method, if any. Each assertion is a comma-separated list of valid Java Boolean expressions, each optionally introduced by a label followed by a colon. The Boolean expressions reference appropriate methods, attributes, or parameters.

```
public interface StackInterface
   extends java.rmi.Remote
{
 int maxSize()
   throws java.rmi.RemoteException;

 int currentSize()
   throws java.rmi.RemoteException;

 invariant currentSize()<=maxSize();

 boolean isEmpty()
   throws java.rmi.RemoteException;

 boolean isFull()
   throws java.rmi.RemoteException;

 void push(Object obj)
   throws java.rmi.RemoteException
   requires obj !=_null, not_full: !isFull()
   ensures not_empty: !isEmpty();

 Object pop()
   throws java.rmi.RemoteExcepiton
   requires not_empty: !isEmpty()
   ensures not_full: !isFull();
}
```

Figure 5. RMI-enabled stack with embedded assertions defined in Biscotti.

Assertion exceptions

We developed a new package, java.lang.assert, to define five new exceptions supporting Biscotti. These exceptions are shown in Figure 6. Precondition, postcondition, and invariant violations cause the program to throw the respective assertion violation exception. Invariant violations throw the InvariantViolation exception, regardless of whether the violation was detected upon entrance to or exit from a method.

The program throws an ExceptionInInvariant exception if an exception occurs inside a Boolean expression that forms an assertion itself. That is, if the program attempts to dereference `null` in an assertion, the assertion-handling code catches the thrown exception (java.lang.NullPointerException) and repackages it inside an ExceptionInInvariant exception.

Assertion reflection

We added the six new interfaces shown in Figure 7 to support assertion reflection. An accessor method, getInvariant(), augments java.lang.Class to report the instance invariant as java.lang.assert.Invariant. Since the current Biscotti implementation does not handle class invariants, support has not yet been defined for class invariant reflection. The getPrecondition() and getPostcondition() accessor methods augment the

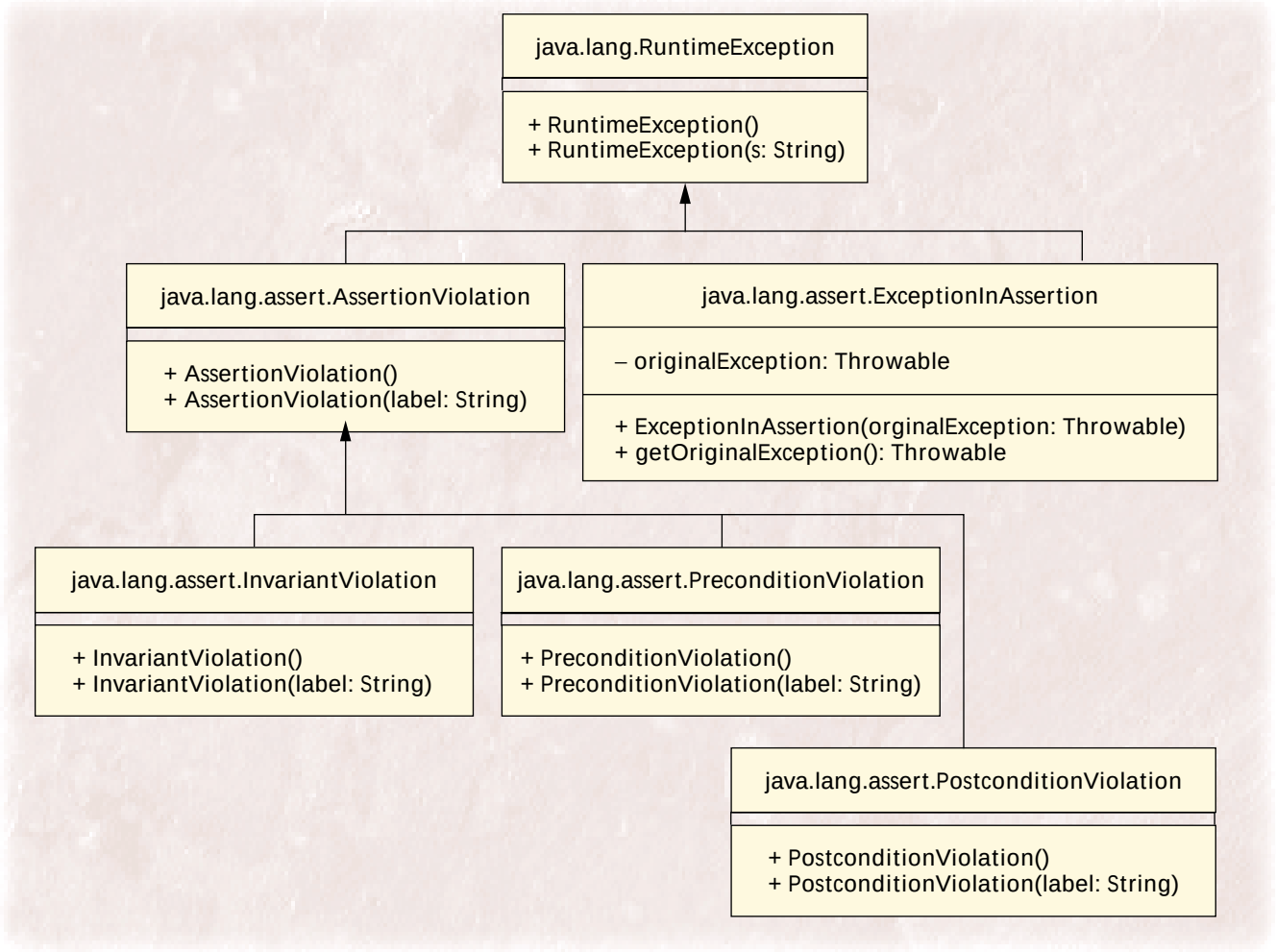

Figure 6. Assertion violation exceptions. Top box: class name; middle box: attributes; bottom box: methods. The plus sign (+) indicates a public attribute or method; the minus sign (–) indicates a private attribute or method.

java.lang.reflect.Method class to report the method's precondition (java.lang.assert.Precondition) and postcondition (java.lang.assert.Postcondition).

Bytecode format

We created a Biscotti compiler by extending the Java 2 SDK, Standard Edition, Source Release, version 1.2 source code. In the prototype implementations, preconditions and postconditions are completely implemented, except for the *old* and *result* keywords, and instance invariants are partially implemented. The compiler makes the assertions available through reflection by adding bytecode information to a class's .class file. To remain compatible with Java code, a class with no assertions has the same bytecode in Biscotti as it does in Java.

For each assertion, the Biscotti compiler adds two methods to the bytecode. The *assertion evaluation method* throws the appropriate exception if the Boolean assertion is false. The *assertion chain method* iteratively calls other assertion chain methods at the level above it until it reaches the root of the inheritance hierarchy.

As in Eiffel, preconditions can only be weakened in subclasses and subinterfaces while postconditions can only be strengthened. Therefore, Biscotti throws a precondition violation exception only if all of the precondition evaluation methods in the inheritance chain throw those types of exceptions; thus, the conditions that a client must satisfy are weaker. In contrast, a postcondition violation exception is thrown if *any* of the postcondition evaluation methods in the inheritance chain throws that type of exception; thus, the conditions that the supplier must satisfy are stronger.

Biscotti invariants are cumulative over inheritance; Biscotti throws an invariant violation exception if any of the invariant evaluation methods in the inheritance chain throws such an exception.

For each assertion, we also add two attributes to the bytecode. One stores the labels associated with the assertion for use in reflection as strings. The other stores a string representing the text of the assertion, also for use in reflection.

The .class file stores the methods and attributes generated for a class's assertions. However, since an interface can't have any nonabstract methods, a static inner class holds the methods and attributes generated for the interface's assertions. A static inner class is a class that is embedded logically within another (outer) class and for which there is only a single instance, regardless of how many instances of the outer class there are. The names of the generated methods, attributes, and inner classes all contain "$" characters to ensure that they can't conflict with any legal names used by a programmer.

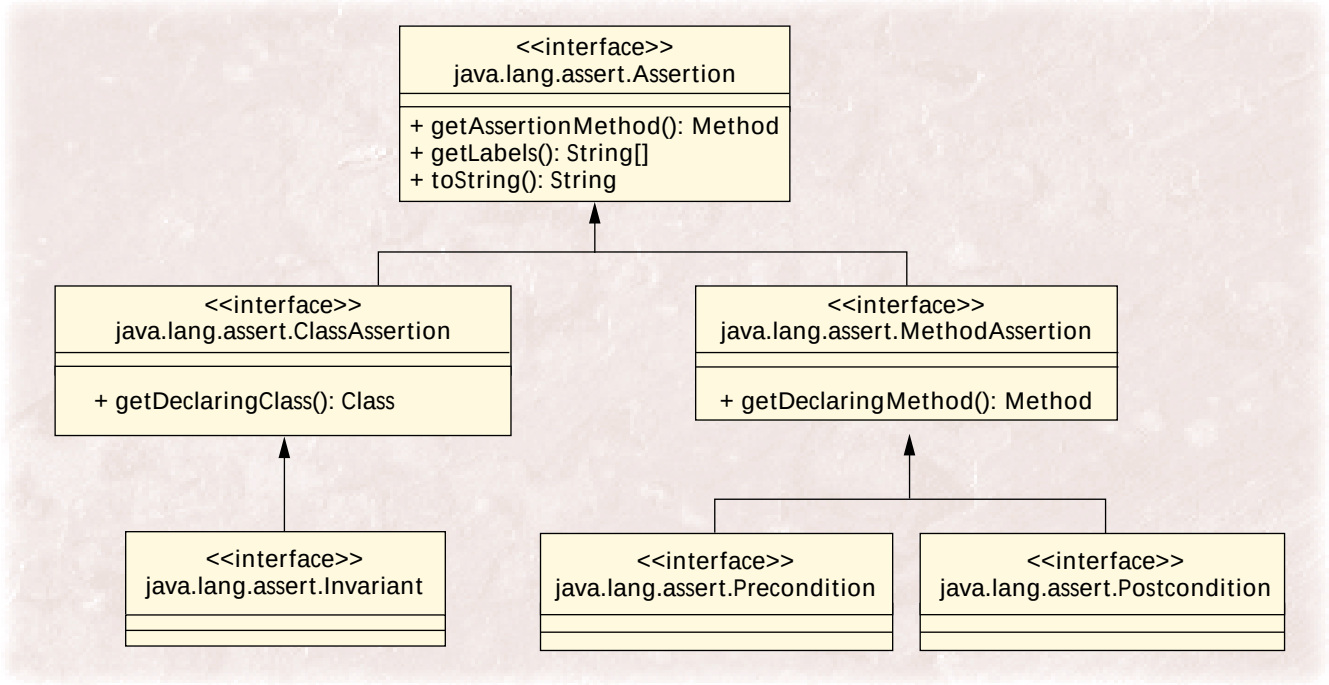

Figure 7. Assertion interfaces.

The Biscotti compiler also instruments the class and interface methods with calls to the appropriate assertion-checking code. Calls to the invariant and precondition chain methods, if these methods exist, are added at the beginning of each method. Calls to the postcondition chain method and invariant method, if these methods exist, are added before each method's return point.

If interface developers are to accept and use behavioral specification, the task must not unduly burden them. Interface developers can naturally transition to using Biscotti because it extends the existing Java language.

We are using Biscotti to further study the implications of adding assertions to distributed software component interfaces. In the course of our work with Biscotti, a number of issues have come to light that did not exist in the implementation of design by contract in Eiffel. Since Java, unlike Eiffel, supports class and instance attributes and methods, it was necessary to differentiate between class and instance invariants. Further, we discovered that, since the nature of distributed software is fundamentally different from that of local, monolithic software, a server developer cannot rely on only well-behaved clients accessing the server. Therefore, it is inadvisable for precondition checking to be disabled in distributed servers. Also, some form of distributed locking and server method synchronization is necessary to ensure that clients operating concurrently do not interfere with each other; in this case, assertions can help detect such interference. In addition, Java's synchronization method must be used for classes with methods that may temporarily violate the class or instance invariant.

A key benefit of this approach is that it is integrated with the target programming language. Thus, programmers can monitor assertions at runtime as part of testing. This will ultimately lead to more robust and maintainable distributed software components. In addition, since assertions are first-class language constructs, visual programming tools will be able to use Java's reflection capabilities to take advantage of embedded assertion information. ❖

References

1. J.-M. Jezequel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997, pp. 129-130.
2. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Upper Saddle River, N.J., 1997.
3. C.M. Holloway and R.W. Butler, "Impediments to Industrial Use of Formal Methods," *Computer*, Apr. 1996, pp. 16-30.
4. K. Finney, "Mathematical Notation in Formal Specification: Too Difficult for the Masses?" *IEEE Trans. Software Eng.*, Feb. 1996, pp. 158-159.

Cynthia Della Torre Cicalese is a principal distributed object engineer at Global Info Tek Inc. She is responsible for the design and development of Java-based distributed systems and has also worked extensively with CORBA. She recently received a DSc in computer science from The George Washington University. She is a member of the IEEE and the ACM. Contact her at cindy@globalinfotek.com.

Shmuel Rotenstreich is a professor in the department of computer science and electrical engineering, The George Washington University. His primary research interests are current techniques in software construction and the interrelationship between economics and finance and computer science. He received a PhD from the University of California, San Diego. Contact him at shmuel@seas.gwu.edu.