EXHIBIT C

White Paper

The Java Language Environment: Contents

Print-friendly Version

CONTENTS | PREV | NEXT

A White Paper

May 1996

James Gosling
Henry McGilton

White Paper

The Java Language Environment

5.1 Dynamic Loading and Binding

The Java language's portable and interpreted nature produces a highly *dynamic* and *dynamically-extensible* system. The Java language was designed to adapt to evolving environments. Classes are linked in as required and can be downloaded from across networks. Incoming code is verified before being passed to the interpreter for execution.

Object-oriented programming has become accepted as a means to solve at least a part of the "software crisis", by assisting encapsulation of data and corresponding procedures, and encouraging reuse of code. Most programmers doing object-oriented development today have adopted C++ as their language of choice. But C++ suffers from a serious problem that impedes its widespread use in the production and distribution of "software ICs". This defect is known as the *fragile superclass problem*.

5.1.1 The Fragile Superclass Problem

This problem arises as a side effect of the way that C++ is usually implemented. Any time you add a new method or a new instance variable to a class, any and all classes that reference that class will require a recompilation, or they'll break. Keeping track of the dependencies between class definitions and their clients has proved to be a fruitful source of programming error in C++, even with the help of "make"-like utilities. The fragile superclass issue is sometimes also referred to as the "constant recompilation problem." You *can* avoid these problems in C++, but with extraordinary difficulty, and doing so effectively means not using any of the language's object-oriented features directly. By avoiding the object-oriented features of C++, developers defeat the goal of re-usable "software ICs".

5.1.2 Solving the Fragile Superclass Problem

The Java language solves the fragile superclass problem in several stages. The Java compiler doesn't compile references down to numeric values--instead, it passes symbolic reference information through to the byte code verifier and the interpreter. The Java interpreter performs final name resolution once, when classes are being linked. Once the name is resolved, the reference is rewritten as a numeric offset, enabling the Java interpreter to run at full speed.

Finally, the storage layout of objects is not determined by the compiler. The layout of objects in memory is deferred to run time and determined by the interpreter. Updated classes with new instance variables or methods can be linked in without affecting existing code.

At the small expense of a name lookup the first time any name is encountered, the Java language eliminates the fragile superclass problem. Java programmers can use object-oriented programming techniques in a much more straightforward fashion without the constant recompilation burden engendered by C++. Libraries can freely add new methods

and instance variables without any effect on their clients. Your life as a programmer is simpler.

5.1.3 Run-Time Representations

Classes in the Java language have a run-time representation. There is a class named `Class`, instances of which contain run-time class definitions. If you're handed an object, you can find out what class it belongs to. In a C or C++ program, you may be handed a pointer to an object, but if you don't know what type of object it is, you have no way to find out. In the Java language, finding out based on the run-time type information is straightforward.

It is also possible to look up the definition of a class given a string containing its name. This means that you can compute a data type name and easily have it dynamically-linked into the running system.

CONTENTS | PREV | NEXT