Techniques for Platform Independence

Juanita J. Ewing and Steve Messick
Instantiations, Inc.
Copyright 1994, Juanita J. Ewing
Derived from Smalltalk Report

Juanita Ewing and

This articles discusses techniques for writing platform independent applications and class libraries. The techniques discussed in this article are useful for modeling environmental changes that affect your application. For example: operating system facilities that vary from platform to platform, windowing libraries for Windows and OS/2 Presentation Manager, a database connection that varies depending on the network configuration, archiving libraries that use either PVCS or Oracle for storage, a color model that depends on the current output device and even Smalltalk platforms such as Smalltalk/V and Objectworks\Smalltalk. These techniques could also be useful as part of a system that models user experience level.

All the techniques in this article, one way or another, are based on polymorphism. They rely on client objects sending messages to platform–dependent objects. The client always sends the same messages, which is where polymorphism comes into play: every platform–dependent object must understand those messages. Thus, during both design and implementation phases, it is important to think about the set of public messages for objects and the requirement for polymorphism.

This article will refer to classes providing platform services as *library classes*, and the client classes that make use of these classes as *application classes*.

Interchangeable classes

Two library classes with the same set of public messages can be used to interface to two different platforms. Because they have the same set of public messages, they are interchangeable. Often, it is convenient to arrange these classes as subclasses of a common superclass because inheritance supports common behavior. The superclass contains common messages, and documents requirements for creating additional subclasses. Often, the superclass is abstract, meaning there are no instances of it. For a discussion on creating abstract classes that are based on similar concrete classes, see the **Smalltalk Report Article Volume 3 Number 2 Abstract Classes by Juanita Ewing.**

The currently appropriate platform–dependent class is known as the *current* class. This is usually a concrete class: a class that can have instances. Because the current class changes, it must not be directly referenced by clients. Let's look at several alternatives for indirectly referencing the current class.

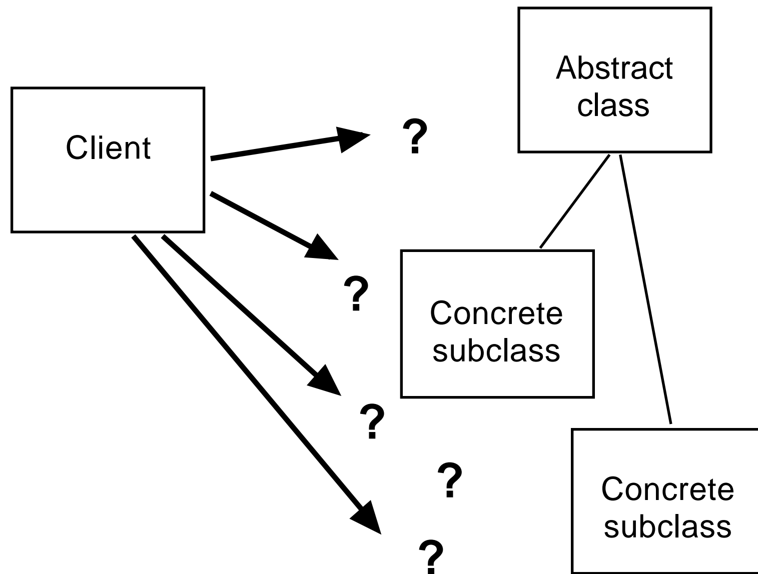


Figure 1. How are library classes referenced?

How do application classes reference the current class? Because the class may change, depending on the environment, application classes cannot reference the current class by name (Figure 1). At compile time the current class is not known. Instead, application classes must reference an indirection to the current class, so that the current class can be replaced.

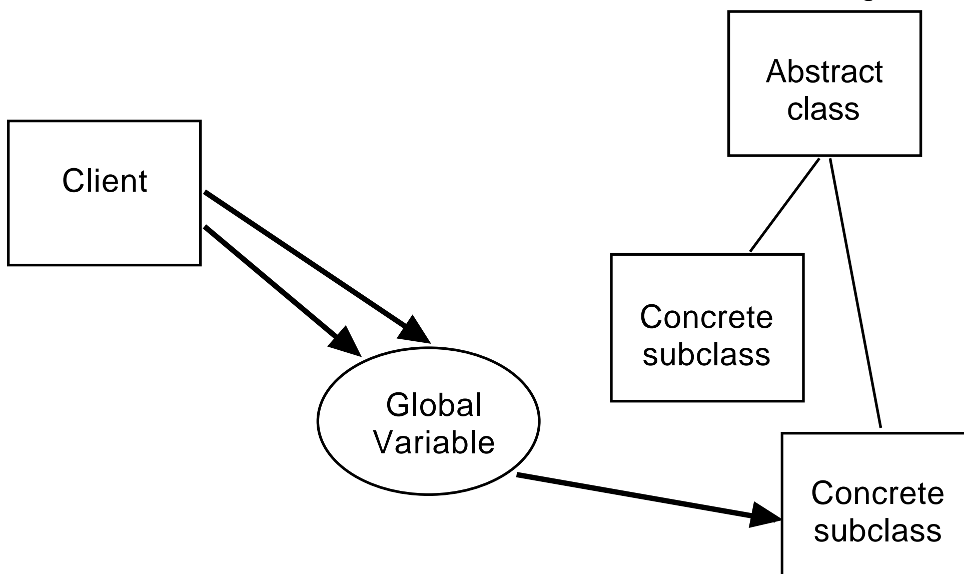


Figure 2. Application code sends messages to a global variable.

Some class libraries use a global variable to refer to the current class (Figure 2). Application code indirectly references the current class with expressions like CurrentDatabaseInterface cancelConnection, which cancels the connection to the current database. In most cases, a variable that is global in scope is not necessary.

An alternate solution is to ask the abstract class for the current subclass (Figure 3). A class instance variable or class variable can be used to hold the actual reference. This solution produces expressions like DatabaseInterface current cancelConnection. A message to the abstract class to retrieve the current class is better than a global variable because

- the functionality is clearly related to the class
- the abstract class is already in the global name scope,
- the abstract class is usually named so that its purpose is obvious to clients, and
- the use of a class message keeps all related functionality in a nice neat bundle that is easier to share and maintain.

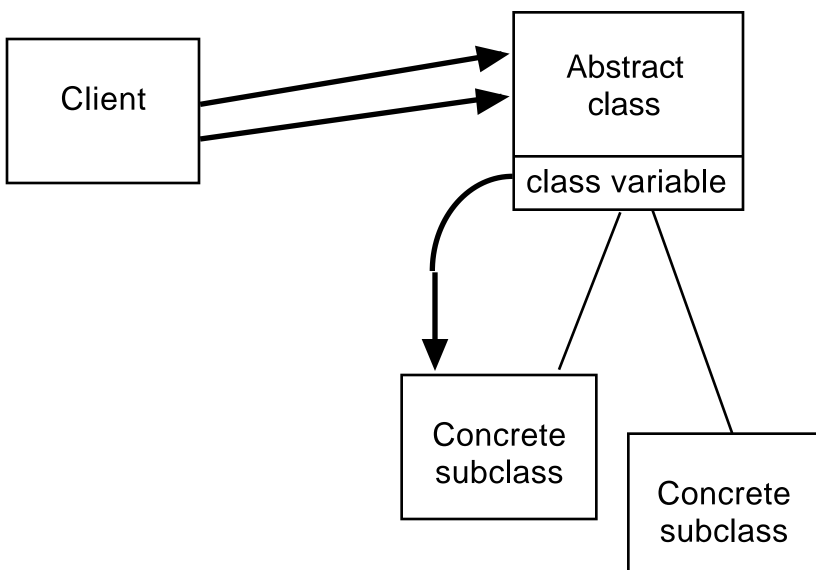


Figure 3. Application code sends messages to the abstract class.

Using interchangeable classes, we discuss three approaches to portability, each appropriate for a different set of assumptions. The approach you select will be based on the specifics of your situation.

Micro–Layering

In the **micro–layer** approach to portability we recognize that the developer must make a portable version of a library class, without changing its public interface. One of our goals is to make the library class itself as portable as possible. The requirement is to extend an existing class to accommodate multiple platforms. We cannot make a new class to replace the existing class without affecting clients of the existing class.

Assumptions for the **micro–layer** approach are:

- library class must be made portable

- clients must not be affected

We can, however, introduce a new non-portable class that isolates the platform dependencies of our library class. Then we rewrite the library class methods to use the non-portable class to perform host-dependent operations. By moving platform-dependent code into a non-portable class we've done two things:

1) eliminated the need to change any public protocol understood by the library class,
2) provided ourselves an easy way to port the non-portable code.

The new class is completely under our control and can be ported by using interchangeable classes, as described above.

As an example, let's assume we're creating a portable Point class that can work with the host's coordinate system, and that it must work on a variety of platforms including Macintosh and OS/2. We immediately see that coordinate system is platform dependent: the x-coordinate increases as we move from left to right on both platforms, but the y-coordinate increases downward on Macintosh and upward on OS/2. To test whether one point is above and to the left of another one we can write for Macintosh:

Point methods
isLeftAndAbove: aPoint

```
    "Return true if the receiver is left and above <aPoint>."
    ^self x < aPoint x and: [self y < aPoint y]
```

And for OS/2 it becomes:

Point methods
isLeftAndAbove: aPoint

```
    "Return true if the receiver is left and above <aPoint>."
    ^self x < aPoint x and: [self y > aPoint y]
```

The different interpretation of y-coordinates occurs throughout the class, and also affects other classes such as Number and Rectangle. These two methods satisfy our second assumption (clients must not be affected) but not the first: separate implementations are required for different platforms. Note that most of the methods are identical; only the test of y-coordinates differs. Isolating the platform-dependent behavior in a new class will make Point (and Number and Rectangle) portable.

Let's introduce a class for coordinate system dependencies, called CoordinateSystem. Using interchangeable classes, we can design a Coordinate System micro layer for Macintosh and OS/2. We'll call the classes MacCoordinateSystem and OS2CoordinateSystem, both subclasses of CoordinateSystem. See Figure 4.

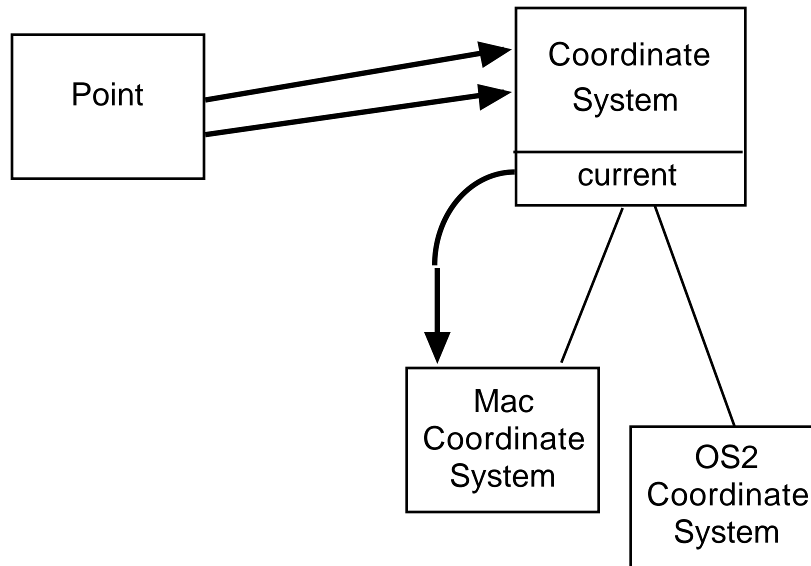


Figure 4. The Coordinate System micro layer.

Since the interpretation of x-coordinates is the same we will define the x-axis protocol in CoordinateSystem. MacCoordinateSystem will interpret increasing y-coordinates downward, OS2CoordinateSystem upward. Rewriting the Point method, we have:

Point methods

isLeftAndAbove: aPoint

```
    "Return true if the receiver is left and above <aPoint>."
    ^(self coordinateSystem is: self x leftOf: aPoint x)
        and: [self coordinateSystem is: self y above: aPoint y]
```

This method is portable, assuming the method coordinateSystem answers an instance of the correct subclass of CoordinateSystem. Additionally, if Point needed to be ported to a platform that interpreted x-coordinates increasing from left to right, it is still portable providing a new subclass of CoordinateSystem is created.

Now let's look at CoordinateSystem and its subclasses. We need to define is:leftOf: and is:above:.

CoordinateSystem methods

is: firstX leftOf: secondX

```
    "Return true if <firstX> is to the left of <secondX>."
    ^firstX < secondX
```

is: firstY above: secondY

```
    "Return true if <firstY> is above <secondY>."
    self implementedBySubclass
```

MacCoordinateSystem methods

is: firstY above: secondY

"Return true if <firstY> is above <secondY>."

^firstY < secondY

OS2CoordinateSystem methods

is: firstY above: secondY

"Return true if <firstY> is above <secondY>."

^firstY > secondY

Macro–Layering

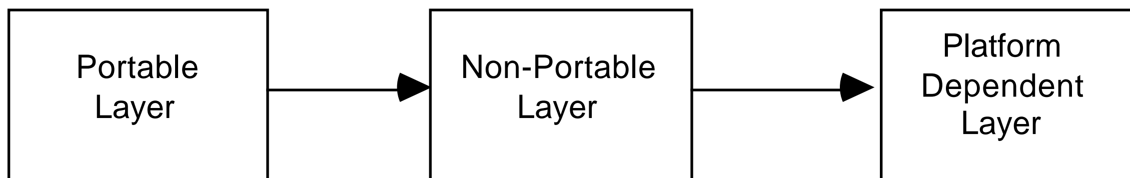
The **micro–layer** approach illustrated the use of interchangeable classes in a microcosm. The next variation applies the same principle on a bigger scale, as the architectural basis of entire systems.

In the **macro–layer** approach, we must make an entire subsystem portable. The assumptions, as in the **micro–layer** approach, are:

- library must be made portable
- clients must not be affected.

This is a good approach to use when developing a portable user interface framework. Digitalk's Smalltalk/V version 2.0 for Macintosh uses it. A different form of it also shows up in ParcPlace's Objectworks\Smalltalk.

The general idea is simple: develop a portable layer that depends upon a non-portable layer for communication with the host platform.



The platform–dependent layer is the service provided by the platform that we need access to in Smalltalk. In may be a user interface like Windows 3.1, a communications toolbox such as Apple's AOCE, or even a third-party product. The only requirement is that it have a well-defined API that can be used by Smalltalk.

The portable layer implements the classes used by client applications. This is the layer most commonly used by Smalltalk programmers. A good example is a user interface framework. The portable classes that implement the framework can be used by application-specific classes to define windows. The application code is protected from platform dependencies, as long as it only uses the portable layer, and is therefore portable.

Interfacing between the portable layer and the platform–dependent layer is the responsibility of the non-portable layer. This layer must do whatever is necessary to transform portable requests, such as create a new window, into the platform specific requests that actually create the window. This often requires transformation of data from a portable representation into the

representation used by the platform, and calling the correct subroutines defined by the platform's API.

To make the system run on another platform, with that platform's implementation of the service, the middle non-portable layer is ported to the new platform. If the portable layer was implemented without relying on any non-portable assumptions then it will work as is. Practically speaking, there may be some code in the portable layer that will not work on the new platform without modification. To ensure portability of client applications the public protocol defined by the portable layer may not change. But applications will work just fine if the public protocol preserves its semantics across platforms, no matter how it is implemented.

This brings us to the issue of specifying the portable layer. This is actually the most difficult part of creating a portable library. Since the underlying service we want to use is itself not portable, we cannot simply look at its API and define our portable protocol in terms of it. We have to create a framework that can be implemented on all potential platforms. The syntax and semantics of the framework has to be specified so that client applications can be defined. The specification must also define the protocol that future extensions to the framework may and may not modify. Also, any methods that have a non-portable implementation must be indicated. To learn more about current research issues in object specification, see the OOPLSA papers by Kiczales and Lamping (OOPSLA ‘92 Proceedings, pg. 435; OOPSLA ‘93 Proceedings, pg. 201).

Let's consider an example. Suppose we are creating the user-interface framework for a family of applications for OS/2 and Macintosh that need to use the host's windowing system. These applications need some "non-standard" windows that always display on top of "standard" windows, sometimes called floating windows or palettes. Looking through the OS/2 manuals we see that this won't be very difficult. OS/2 provides the capability we need. However, extremely careful reading of Inside Macintosh reveals that we may be able to get one window of this sort, but if we need more than one (and we do) then we're out of luck. It turns out that we have to reimplement a portion of the Macintosh window manager class to solve this problem.

By applying the principle of interchangeable classes to the problem description, we can design and specify a WindowManager class that has implementations for OS/2 and Macintosh. In our portable user interface library we define the classes StandardWindow and FloatingWindow to implement the two varieties of window we need. These classes use WindowManager to create and destroy windows and to make windows visible or invisible. We'll also have non-portable classes, OS2Window and MacWindow, to implement the platform-specific window functions like setting window title and size. The result is a user-interface framework for building portable applications, and a framework that is itself largely portable. The design includes no inherent performance penalty for either platform.

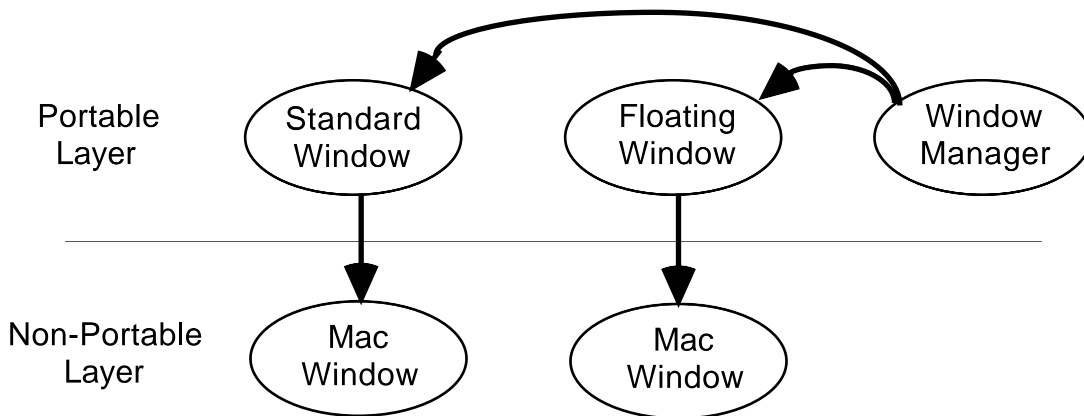


Figure 4. The major objects required to represent one standard window and one floating window on a Macintosh.

If, on the other hand, we had tried to design the framework based only on the OS/2 API we probably would have arrived at a much less portable version of the framework. It is quite likely that our design would not have included either WindowManager or FloatingWindow. After all, why should it? OS/2 takes care of all the bookkeeping required. We would, of course, have StandardWindow and OS2Window because we're using the layering method to isolate platform dependencies. But that alone is not enough to ensure portability. If missing functionality must be implemented for some platform, then the design must allow for that. If the functionality is not part of the design, client applications will be based on a sub-optimal design and we will be faced with enormous backward-compatibility problems. Rather than designing a system based on the functionality available on a platform, we design the system to meet our requirements.

An interesting variation on the layering theme is found in Objectworks\Smalltalk. The non-portable layer is implemented in the virtual machine. The portable layer is implemented in Smalltalk; it is entirely portable because all platform dependencies are hidden in the virtual machine. Using this approach, Objectworks ensures portability of the applications defined in Objectworks\Smalltalk and also of their image file.

Platform server
The last variation is a pragmatic approach that is often used to extend the set of platforms an existing application can support. In this approach, a platform server class is used to contain all platform specific code that the application relies on. There is one platform server class for each platform, providing a consistent interface to platform functionality.

This can be used when an application relies on two platform libraries that do not have an identical public interface. Our advice is to use this technique only if you do NOT have control of library classes, or as a stop gap measure if you can rewrite library classes. If possible, you should refactor and expand the set of library classes, resulting in many interchangeable classes.

Assumptions for the **platform server** approach are:

- many small variations in library classes

- developer cannot rewrite library classes

For this technique, let’s discuss an example involving the platforms Smalltalk Agents for Macintosh, and Objectworks\Smalltalk. Suppose we have an application that must run on Objectworks\Smalltalk and on Smalltalk Agents for Macintosh. This application requires streams and a collection that holds its elements in order. We also want the ability to do some rudimentary performance analysis, and therefore need an operation that can be used to time the execution of a block.

The way we access the required functionality is different with each Smalltalk platform. In order to isolate the bulk of our application from platform dependencies, we compartmentalize the variations for each platform into a platform server class. The class ObjectworksServer is a mapping to functionality on the Objectworks\Smalltalk platform, and the class SmalltalkAgentsServer is a mapping to functionality on the Smalltalk Agents platform.

Let's examine a sample of methods from the server classes. Methods that identify an appropriate class, such as the method orderedListClass, are useful when two platforms have similar classes with different names. It can also be useful to help identify dependencies and collaborations.

Ordinarily, when operating on a object, we send messages directly to the object. When we provide a mapping to that functionality in a platform server class, we must send a message to the server class. The original object becomes an argument. The message readFrom:through: provides an operation on a stream, but it is a message sent to the server class with the stream as an argument.

Sometimes the server class will end up implementing functionality that is simply not present on one of the platforms. The message timeToExecute: is a mapping to existing functionality for Objectworks\Smalltalk, but new functionality for Smalltalk Agents.

****Format suggestion - these two lists, one for Objectworks and one for Agents, would be nice side by side****

ObjectworksServer methods

orderedListClass

```
	“Return a class that holds its elements in order.”

	^OrderedCollection
```

readFrom: aStream through: anObject

```
	“Return a collection of elements read from <aStream>, starting from the current stream
position up to and including <anObject>.”

	^aStream through: anObject
```

timeToExecute: aBlock

```
    "Return the number of milliseconds to execute <aBlock>."

    ^Time millisecondsToRun: aBlock
```

SmalltalkAgentsServer methods

orderedListClass

```
    "Return a class that holds its elements in order."

    ^List
```

readFrom: aStream through: anObject

```
    "Return a collection of elements read from <aStream>, starting from the current stream
position up to and including <anObject>.'

    | throughCollection |
    throughCollection := aStream upTo: anObject.
    throughCollection add: aStream next.
    ^throughCollection
```

timeToExecute: aBlock

```
    "Return the number of milliseconds required to evaluate <aBlock>, rounded to the
nearest ms. The computation is at best approximate because the basic unit provided by Apple
is a tick (1/60 sec)."

    | timer startTime |
    timer := ClockDevice new.
    startTime := timer ticks.
    aBlock value.
    ^timer ticks - startTime * 100 + 3 // 6
```

Dynamic vs. Static

There is another issue, orthogonal to variations on interchangeable classes, that deserves discussion. This is the issue of how applications can be configured.

If an application runs on one platform at a time, developers can use configuration management tools to build their application with the appropriate platform–dependent classes. The result is several versions of an application, one for each platform. We call this situation a static configuration, and do not discuss it in detail. There are several commercially available tools for configuration management of Smalltalk applications such as Team/V and ENVY Developer.

If the application must run on multiple platforms, developers can design their application to dynamically support the appropriate platform. In this situation, called a dynamic configuration, the result is one version of the application that includes all platform–dependent classes.

Setting the Current Class

There are several different ways of installing the current platform–dependent class. The exact mechanism depends on how often the environment changes. Does the current class potentially change every time it is accessed, or does the default change less frequently?

Some classes are installed when Smalltalk is started. In Smalltalk/V for Macintosh, any object can register for notification when Smalltalk starts with an expression like this:

```
SessionModel current
        when: #startup
        send: #setCurrent
        to: PlatformInterface
```

The setCurrent method includes an expression to set the current class. We use the class ServiceRegistry to identify the current platform. The method setCurrent is implemented by the class PlatformInterface.

setCurrent

```
"Set the current platform interface class based on the current platform."

| platformName |
platformName := ServiceRegistry globalRegistry
                        serviceNamed: #PlatformName
                        ifNone: [^self installStub].
platformName = 'Macintosh'
        ifTrue: [^self current: MacPlatformInterface new].
platformName = 'OS/2'
        ifTrue: [^self current: OS2PlatformInterface new].
platformName = 'Windows'
        ifTrue: [^self current: WindowsPlatformInterface new]
```

Another strategy is to wait until a current class is requested and then determine the current class if necessary.

current

```
"Answer the interface used by the current platform."

Current == nil
        ifTrue: [self setCurrent].
^Current
```

Even with this strategy, the old current class must be flushed at some appropriate time, such as image startup, so that the current class will be installed when the class is accessed. This expression should be executed sometime during application startup:

```
PlatformInterface flushCurrent
```

Conclusion

Developing platform independent applications means more than writing code for different hardware platforms such as Macintosh and IBM PC's. Different software platforms can also be addressed with the same techniques.

Carefully consider all possibilities for extension of your application while choosing design and implementation techniques. If you follow the approaches laid out in the article, it will be much easier to move your application from one platform to another. There are, no doubt, other interesting techniques that can be used to ease the task of porting between platforms. We'd like to hear about them. Send descriptions of your own practices to juanita@digitalk.com.