Automated Software Transformation & Translation From Legacy Code to Java

By Florin Mihaila and Tom Holmes

Most businesses today face the challenges of developing, maintaining and enhancing applications that are developed using older function-deficient technology. This technology is the foundation of applications purchased to help drive their core business, or used in applications developed internally, or used in applications created for resale.

Many such applications can be found on AS/400 and iSeries platforms. These older development technologies have limitations that impact the functionality for both the customers using these applications as well as the ISVs (Independent Software Vendors) developing applications for resale.

One of the most popular alternatives to legacy development tools and applications is Java, a current technology that solves many of the problems being experienced by customers using these legacy platforms.

Java is an open, flexible and secure development environment that provides many advantages, including:

- Platform independence for applications.
- Component-based design, allowing for easy future maintenance and enhancement of applications.
- Open, standards-based architecture.
- Availability of industry-wide support and compatible components.
- Scalability, allowing applications to grow and expand along with corporate strategies.

Migrating platforms and converting applications from any legacy language to Java is no small task. Without the use of sophisticated automated tools such conversion projects would require manual coding that could take many months to complete and in fact can easily escalate to multiple person/years of manual work. Many projects will not be converted if a re-write is the only option.

What are the issues driving the consideration of transformation from the legacy system to Java?

Staff Limitations – The most highly skilled individuals are moving to newer, more modern technologies that offer them greater challenges, rewards and opportunities. Companies are losing critical business knowledge when these resources move on.

Architectural Limitations – Companies are reluctant to tie themselves to proprietary technology. Customers seek open standards, and flexible solutions such as J2EE. Additionally, reliance on a sole vendor ties customers to the market and product fluctuations experienced by that vendor, impacting future products, support and available resources. Platform and vendor independence form an insurance policy against the inevitable fluctuations in the technology marketplace.

Application Development Tools Vendor Consolidation – The current consolidation in the database marketplace is also impacting application development tools. For all the same reasons, as well as cost and efficiency reasons, companies want to standardize on as few development platforms as possible.

Non-Strategic Technology Platform – Many ISVs (independent software vendors) are experiencing difficulty selling and maintaining applications that are written in these legacy languages. ISVs need to utilize modern technologies that help them react quickly to the changing business needs experienced by their customers.

Limited Support – Some legacy vendors are unwilling or unable to keep their products current, often-times not offering support for the latest market developments. For example some vendors took more than a year to enable their products to work with Windows XP.

Transformation Options

1. **Do Nothing**

While this appears to be a conservative and workable solution, it is the most risky alternative if these are active applications. Functionally these applications will continue to fall behind, or have awkward enhancements.

2. **Black Box**

If the applications do not need to be maintained, there are "wrapper" technologies available that will support treating the application as a set of inputs and outputs.

3. **Convert to Packaged Applications**

Corporations can choose to replace their custom applications with packaged applications. Often the internal processes and procedures will need to be modified to comply with the application, or the application will need to be modified to conform to the existing work methods, or both. This significantly increases the risk involved.

4. Rewrite

Companies can choose to rewrite the application, using more modern technology. This is the longest and most expensive approach. It has the same risks as any application development effort, plus will significantly extend the time before the application will be available.

5. Migrate/Convert

Another approach is to convert using modern automation tools, to minimize the turnaround time, and the risk. These tools provide accurate, reliable, and repeatable conversions.

This article and attached diagram deal with two areas that are especially interesting in the context of automated conversions from PowerBuilder to Java:

- Swing GUI classes
- DataWindows

If you are relatively new to Java, you'll find this information more easily understood if you first refer to the Farr/Coulthard article in the November '97 issue of the TUG magazine and to Mark Orlan's article on the TUG website "The e-business opportunity for AS400 Shops".

I. Swing GUI Classes

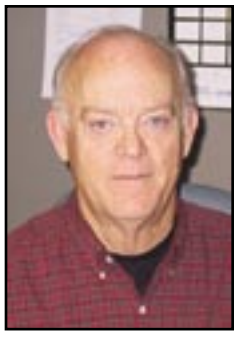
Here we discuss our approach to the design of the Java code generated by JavaConvert/PB (Java translator) and the requirements that this imposes on the design of our Java runtime library, called PBJ. We discuss the problems encountered and the choices we made in light of our design goals. One approach to PowerBuilder-to-Java code translation is to attempt to model each PowerBuilder visual type directly as a visual Java class, such as a Swing class. In the following, we analyze the problem in more detail, and explain why this approach does not work and why we chose to translate to a peer model instead.

The majority of top-level types defined in a typical PowerBuilder application are visual containers. Examples include regular windows (corresponding to top-level frames in Java), and custom user objects. Instances of these types act as containers for other components, such as buttons, treeview controls, and edit fields.

PowerBuilder enforces a strict correspondence between the visual containment hierarchy and the type structure of the program. For each component residing inside a visual container, PowerBuilder defines an inner type inside the type of the container. Exactly one instance variable of each inner type is declared, typically having the same name as the type (this is allowed in PowerScript.) Each of the inner types can have event handlers and other methods attached.

Florin Mihaila

About the Authors: ***Florin Mihaila*** *is Systems Architect and Project Lead for the PowerBuilder to Java (JavaConvert/PB) project at Techne Knowledge Systems. He can be reached at fm@techne.ca.*
Tom Holmes *is Operations Officer for Techne Knowledge Systems and can be reached at tom.holmes@techne.ca*



Tom Holmes

Since Java directly supports inner classes, the most natural translation is to transform each PowerBuilder type into a Java class: top-level PowerBuilder types become top-level Java classes, and inner PowerBuilder types become inner Java classes. However, many GUI designers do not support visual editing of components implemented as inner classes, rendering this approach unacceptable for most customers.

There are other reasons why it is a bad idea to translate PowerBuilder types directly into Swing classes. For example, in PowerBuilder, a window class can be instantiated in more than one way in the same application. The programmer may open a first instance as a top-level window, and a second instance as an internal window inside an MDI (Multiple Document Interface.) frame. Both instances belong to the same PowerBuilder class.

In Java, however, this is not possible. The programmer must use one Swing class for top-level windows (JFrame), and another Swing class for internal MDI windows (JInternalFrame.) The two classes are not related, and since Java doesn't support multiple inheritance, we cannot support the above functionality by translating the PowerBuilder window class directly into a Swing-derived class. Using composition instead of inheritance can solve all of these problems. This allows us to decouple the PowerBuilder-imposed type hierarchy from the Swing-imposed one. This is achieved by using a peer model, where each PowerBuilder type is always implemented as a non-visual Java class, which in turn delegates all the visual responsibilities to a Swing peer.

In addition to resolving the conflicts enumerated above, this scheme makes it easier to re-target PBJ to another GUI widget toolkit, such SWT (included in the IBM's Eclipse framework.) The Java code is therefore split into high-level code, containing all the automatically translated business logic, and low-level GUI-specific code, automatically generated as a visual class (currently Swing.)

The high-level code doesn't have any type dependencies on any specific low-level GUI classes. Instead, the access to the GUI peers is mediated by a set of Java interfaces. Actual peers are created at runtime via a PeerFactory, which instantiates objects from a particular peer back-end (Swing, SWT, etc.) according to a static library setting.

This scheme allows both the generated code and the library code to satisfy the following requirements:

- A hierarchy of Java interfaces mirrors the PowerBuilder type hierarchy from the original application. These interfaces declare all the properties, events, and functions supported by the original PowerBuilder classes. This is needed in order to support PowerBuilder code that relies on runtime type information (for instance, code that inquires about the type of a specific instance.) Such code can be translated into equivalent Java code, using the reflection capabilities of Java;
- Application code is always translated to non-Swing classes, which implement the above interfaces. They inherit from base non-Swing classes provided by the PBJ runtime library;
- The visual peers are Swing objects that implement the peer interfaces accessed from the high-level classes described above. They are derived

Java Code Structure Example

Assumes translation of the following PowerBuilder types:

- `MyWindow from window`
- `MyUserObject within MyWindow from userobject`

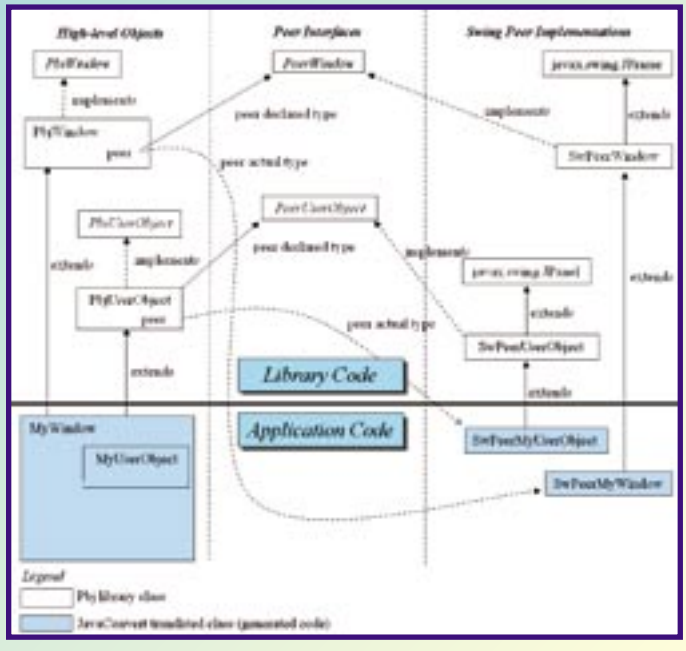


Figure 1.

from standard Swing objects, follow the usual Swing programming conventions, and can be manipulated in standard IDE visual editors.

One side-effect of this design is that the number of Java classes required to translate all visual PowerBuilder types from an application doubles: each PowerBuilder visual type translates into one high-level class, and one GUI-specific peer class. All the PowerBuilder business logic, implemented as custom event handlers and regular methods, is translated into Java methods belonging to the high-level classes. The peer classes contain only the Swing definition of the visual objects, and don't contain any business logic code. We believe the increase in the number of classes is more than compensated for by the advantages gained.

II. DataWindows

In a nutshell, DataWindows are data-aware GUI components. They are components, because they can be reused in many different contexts in an application, or across multiple applications. They are GUI components because they provide advanced presentation capabilities. Finally they are data-aware because they are configured with table names, field names, field types, field precision, SELECT statements, and other database-related information describing the data handling capabilities of that specific component. The DataWindow construct is very powerful. For this reason, DataWindows are used as the main building blocks for most PowerBuilder applications. Since the business logic of so many existing systems revolves around DataWindows, it follows that any migration solution for PowerBuilder applications must solve the problem of DataWindow conversion.

The PBJ runtime library implements the DataWindow concept and makes it available to translated applications. Sophisticated features, like data buffering, batch database access, complex data grouping, filtering, sorting, data sharing, multiple presentation styles, and advanced reporting capabilities, are all provided by the PBJ runtime.

Each DataWindow from an existing PowerBuilder application is translated into two Java classes following the peer model described earlier: a high-level class, and a low-level GUI class (the peer). The high-level class inherits directly from the PBJ class responsible for implementing the core DataWindow mechanisms in Java. The GUI peer is derived from a PBJ peer class, which in turn is derived from JPanel (a standard Swing class.) This allows the programmer to inspect and change the properties of translated DataWindows inside a visual GUI editor. See figure 1 for graphic description.

We have discussed just two aspects of automated transformation and the associated challenges inherent in delivering code that accurately reflects the legacy code business logic. As you can see from the class diagram (**Figure 1**), there are many other "interesting" areas that could be discussed in depth. TUG