

## The Structure of the Code Generated by JConvert/PB

All PowerBuilder applications, at their lowest level, directly depend on the core PowerBuilder API. This API is composed of many predefined global functions, along with a rich class hierarchy, providing miriads of services. The total number of global functions, methods, properties, and events defined by the PowerBuilder API exceeds 3000.

The majority of these services do not exist in the standard Java SDK. The PBJ runtime library is a Java library which provides the missing services, and which is functionally equivalent to the PowerBuilder runtime library.

One design approach for such a library is to attempt to model each PowerBuilder visual type directly as a Java Swing class. However, this direct approach does not work, for the reasons explained below.

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, edit fields, and so on.

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 the PowerScript language.) Each of the inner types can have event handlers and other methods attached.

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 frame (MDI stands for Multiple Document Interface.) Both instances belong to the same PowerBuilder class.

In Swing, 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.

All of the above problems can be solved by using composition instead of inheritance, allowing us to decouple the PowerBuilder-imposed type hierarchy from the Swing-imposed one. This is achieved by using a peer model, where each PowerBuilder visual type is implemented as a non-visual Java class, which in turn delegates all the visual responsibilities to a Swing peer. The translated code is therefore split into "high-

level” code, containing all the business logic, and “low-level” GUI-specific code, automatically generated as a visual Java 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 peer interfaces. Actual peers are created at runtime via a PeerFactory, which instantiates objects from a particular peer back-end (Swing, in this case, but see below.) according to a static library setting.

In conclusion, the Java code has the following properties:

- In the PBJ library, a hierarchy of Java interfaces mirrors the PowerBuilder type hierarchy. 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 (e.g. 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 implementing the above interfaces. They extend base non-Swing Java classes provided by the PBJ runtime library;
- In the case of both the PBJ library code, and the translated application code, the visual peers are Swing classes implementing peer interfaces accessed from the high-level classes described above. They are derived from standard Swing objects, following the usual Swing programming conventions, and can be manipulated in standard IDE visual editors.

## Java Code Structure (Swing client/server)

Assumes translation of the following PowerBuilder types:

- MyWindow from window
- MyUserObject within MyWindow from userobject

