The Platform-Blending Concept

When In Rome, Do as the Romans Do…

Abstract

Creating software in Java and hence making it platform independent[1] is only half the battle. To make a software application truly platform independent, the application must behave as any other platform specific application would behave on that platform. Otherwise, the platform independent application sticks out like a "sore thumb". Thus platform independence is only the first step to a larger, ultimate, and more fulfilling concept of "*platform-blending*".

This article describes the process of transforming the Precision Choice™ Core Engine components into Windows NT services that allowed Online Insight to achieve platform blending on Windows NT. Although, this article uses Windows NT as an example platform to illustrate the platform-blending concept, the ideas presented within are equally valid on any other platform.

The Need for Platform Blending

Java has gained widespread acceptance as a language for creating platform independent software applications. However, even JavaSoft, the creator of Java realized that platform independence, although attractive to software development companies, is not enough to compel customers to buy such software. Hence, the Java Abstract Windowing Toolkit (AWT), which provides user interface capability to Java programs, was quickly followed by the Java Foundation Class (JFC) library for creating user interfaces. Unlike the AWT, the JFC library allows the look and feel of the user interface to be adjusted to the platform on which the application is being run. A variety of look and feel options such as Microsoft Windows, X-Motif, etc. are available via the JFC. However, look and feel is just one component of the platform-blending concept. Platform blending includes every aspect of the software application on a specific platform, from look and feel to installation, to startup and shutdown.

Following software industry best practices, Online Insight has utilized a component driven approach for developing Precision Choice. At the highest level these components appear as layers in the software, which are namely the User Interface, the Presentation Services, and the Core Engine layer. Each layer itself is composed of components as well. For example, the Core Engine layer is composed of an Adaptor, Command Processor(s) and Datastore(s) components.

[1] Software written in Java needs to be thoroughly tested on each platform that is claimed to be supported.

Each one of these is further composed of components. This approach provides Online Insight maximum flexibility in keeping up with rapidly advancing technology and customer needs.

A service on Windows NT is an executable object that is installed in a registry database maintained by the Service Control Manager (SCM), which is an integral part of Windows NT. The executable object associated with a service can be started at system boot time by a boot program or by the system itself, or it can be started on demand by the SCM. Windows NT has two types of services, namely a Win32 service and a driver service. A Win32 service is a service that conforms to the interface rules of the SCM. This enables the SCM to start the service at system startup or on demand and enables communication between the service and service control programs. A Win32 service can execute in its own process, or it can share a process with other Win32 services. A driver service is a service that follows the device driver protocols for Windows NT rather than using the SCM interface.

Windows NT services allow applications that provide a “*service*” to other programs/users to start-up during system boot-up without the need for a user to log on to the machine or manually start the application. Services are needed to perform user-independent tasks such as directory replication, process monitoring, or services to other machines on a network, such as support for the Internet HTTP protocol. All major applications on Windows NT that provide such “*services*” are available as a Win32 service. Thus for Precision Choice to be truly accepted by Windows NT users the logical course of action was to make the Core Engine components (the “*service*” providers) available as Win32 services.

The Approach

Online Insight could have re-written all the Core Engine Components in C++ in the form of Win32 services. However, this is not what platform blending is about because by doing this the software application has just lost its platform independence, which is an integral part of the platform-blending concept. In addition, this would not have been the most efficient/reusable way to accomplish the task. Figure 1 shows the architecture used

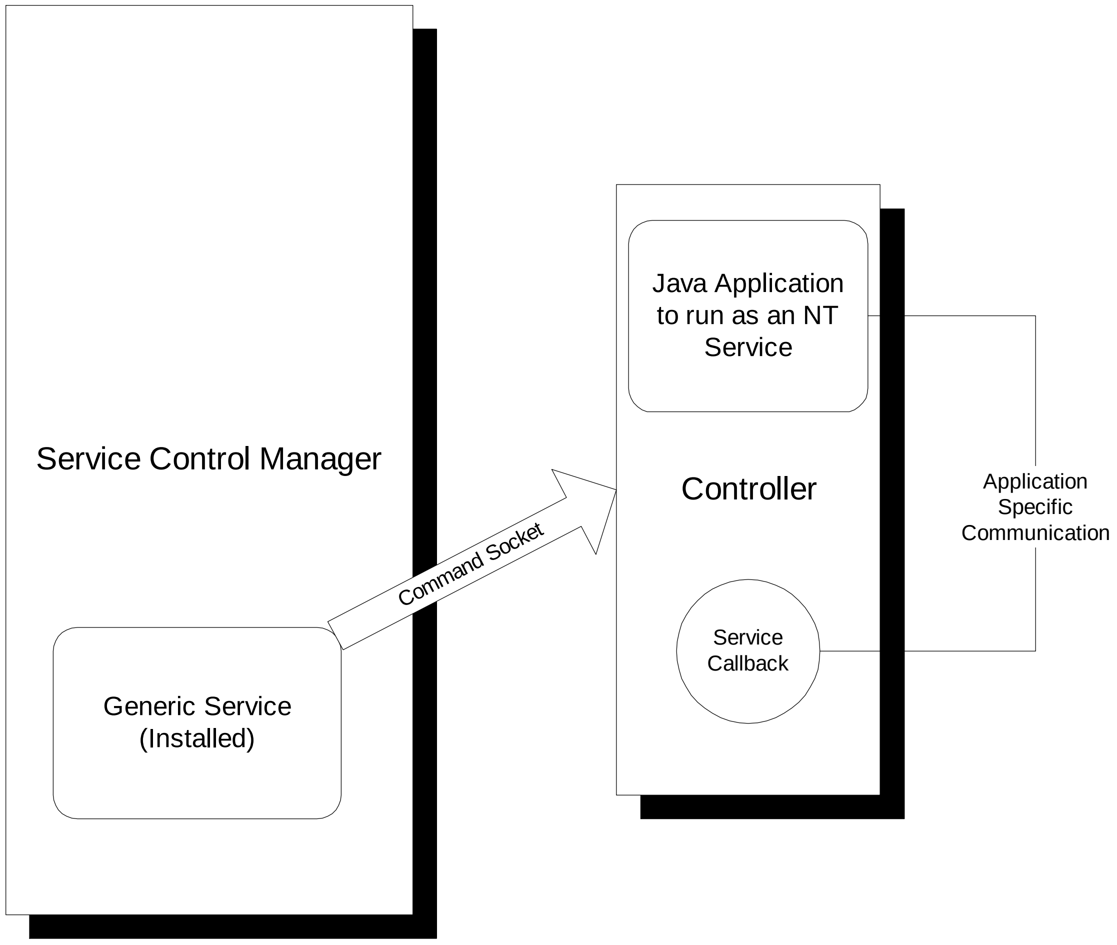
Service Control Manager
Generic Service (Installed)
Command Socket
Java Application to run as an NT Service
Controller
Service Callback
Application Specific Communication

Figure 1: The Basic Architecture

The architecture has introduced three new components, which wrap[2] the Java application that is to be made available as an NT Service. Only the Generic Service component needs to be installed. The components are described below:

1. The Generic Service

The Generic Service is a general purpose NT Service written to conform to the NT Win32 service protocol. This component is written using C++. Each Java application that is to be made available as a Win32 service must have its own copy of this Generic Service component, which is simply a matter of copying the executable and renaming it to the name of the service desired. For example, to install a service named "AcmeService", copy and rename "GenericService.exe" to "AcmeService.exe".

To install this service, from a command prompt type the following:

> *AcmeService –i <controller.properties> <communication port> [JVM parameters] [dependencies…*
> *]*

where,

-i:
informs the service to install itself

<controller.properties>:
The absolute path of the properties file for the Controller component (See the Controller for details)

<communication port>:
The port used by the service to communicate with the Controller. This port must match the port specified in the controller properties file.

[JVM parameters]:
Arguments for configuring the Java Virtual Machine (JVM). This is an optional parameter. If there are multiple parameters enclose the whole list in double quotes.

[dependencies…]:
A list of space separated services (their names), all of which must start before this service starts. Specifying these services will ensure that these services start before the installed service. This is another optional parameter.

The service can be uninstalled as follows:

> *AcmeService –u*

where,

-u:
informs the service to uninstall itself.

The version of the service itself and seeing if the service is installed can be done as follows

> *AcmeService –v*

where,

[2] Warp or decorate, See the "decorator" design pattern for details about this concept.

-v:
informs the service print the above information about itself.

The Generic service will start up the Controller and pass the Controller the properties file specified as the <controller.properties> parameter during the install. Thereafter, the Generic Service will then send commands to the Controller via the communications socket specified during the install. For each discrete command, the Generic Service must open up a new socket connection to the Controller. This is because the Controller closes the socket connection after each command.[3] The communication protocol between the Generic Service and the Controller is extremely simple and is "home-grown". It essentially consists of a byte stream that propagates an action (such as "start", "stop", "pause", and "resume") initiated on the service from an application such as the SCM to the Controller component so that the Java application may respond appropriately (See the Service Callback component for details).

2. The Controller

The Controller is a Java component that is responsible for starting up the Java Application that is to be made available as a Win32 service. This application is specified in the Properties file for the Controller (which is passed to the Controller by the Generic Service and corresponds to the <controller.properties> parameter during the service install). The Controller will pass this application a properties file that is specified inside the Controller properties file. This properties file informs the application about parameters specific to itself. The Controller also opens up a socket on the port specified in the Controller properties file and will listen for commands from the Generic Service on this socket. Once it receives a command, it will call the appropriate method on the Service Callback class and close the socket connection to the Generic Server[4].

3. The Service Callback

This is a Java class that must implement the *com.onlineinsight.intercomponent.windows.Service* interface. This interface contains methods, which are called by the Controller in response to commands sent by the Generic Service via the SCM, or any other means, which correspond to commands to "pause", "resume", and "stop" the service. Since the action taken for each Java application in response to these commands is application specific, this callback class is required to be provided by the application provider. However, if no special action is required for the application in response to these commands, the application provider may use the "null" callback class provided (which is *com.onlineinsight.intercomponent.windows.NullCallback*). This has been done in the case of the CommandProcessor component of the Core Engine, which does not have a special command response requirement. However, the Adaptor and Datastore components of the Core Engine do have special (specific) requirements and hence a callback class specific to each was created. Each method of the callback class receives the properties that were passed to the Java application when it was started and also the arguments received by the Controller when the Generic Service started it. Using these the callback class can communicate with the Java Application in the appropriate (application specific) way.

[3] Sort of like HTTP, which indicates a stateless, connectionless protocol.
[4] This refers to the client connection. The Controller never closes the communication socket port.

The Service Choreography

Figure 2 shows a sequence diagram highlighting the major flow of actions in the architecture.

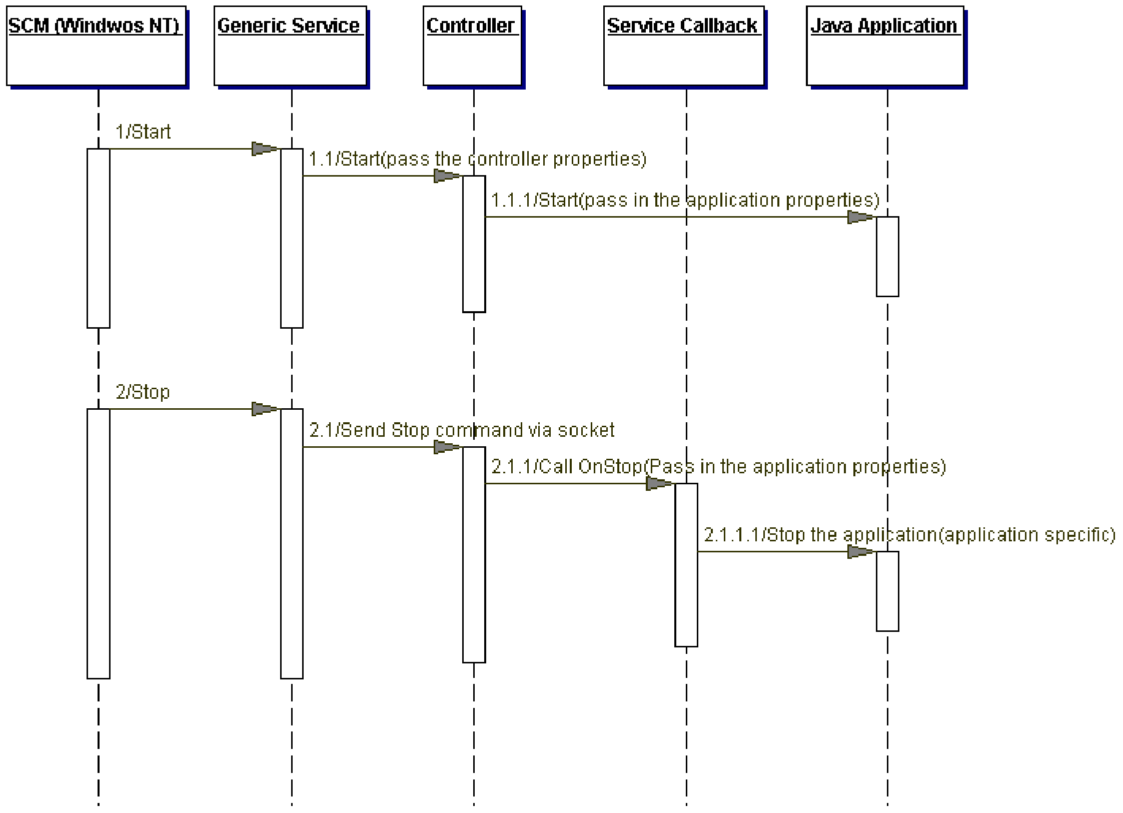
SCM (Windwos NT)
Generic Service
Controller
Service Callback
Java Application
1/Start
1.1/Start(pass the controller properties)
1.1.1/Start(pass in the application properties)
2/Stop
2.1/Send Stop command via socket
2.1.1/Call OnStop(Pass in the application properties)
2.1.1.1/Stop the application(application specific)

Figure 2: The Basic Flow

Conclusion

Platform Independence is a very interesting and useful concept, which software professionals have strived to achieve since the dawn of computing. Java takes us a long way in achieving this. However, as discussed in this article, platform independence yields maximum benefit to its creator when it is taken a step further to what this article has described as “platform blending”. Platform blending and platform independence are not mutually exclusive. Rather, platform independence is an essential subset of platform blending. Using proper software design principles[5], platform independent applications can easily be transformed into platform blended applications without any change to the application itself or losing the platform independence of the application. Such platform-blended applications greatly enhance the value derived by customers in all interactions with the application. Greater value means a happier customer, which is ultimately what business (and software development) is all about.

[5] Such as Design Patterns.