The Intel® VTune™ Performance Analyzer: Insights into Converting a GUI from Windows* to Eclipse*

Aaron Levinson
Intel Corporation

Copyright © 2004, Intel Corporation.
All rights reserved.

Intel, VTune and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries.

Introduction

- The Intel® VTune™ Performance Analyzer (a mostly C++-based product): current product offerings
  - VTune analyzer for Microsoft* Windows (standalone, Visual Studio* .NET*, and command line user interfaces). Graphical user interfaces implemented mostly in MFC (Microsoft Foundation Classes).
  - VTune analyzer for Linux* (command line user interface)
- Next step: develop a GUI for the VTune analyzer on Linux
- Ultimately, Eclipse* was chosen for Linux:
  - Fast track to an integrated tool suite
  - Cross-platform
  - Licensing

intel®

*Other names and brands may be claimed as the property of others.

Considerations: Windows* to Eclipse*

- GUI has to be implemented in Java*
- Logic (non-GUI, base functionality) C++ code uses JNI (Java Native Interface) to communicate with GUI (and vice versa)
- Not a straight port of Microsoft Windows* GUI concepts to Eclipse. Important to use Eclipse user interface conventions
- Important to keep logic code as OS-independent as possible.

intel®

*Other names and brands may be claimed as the property of others.

JNI and Eclipse*

- To keep implementation as simple as possible, use one native library from Java* side and make all standard calls to C++ side through this library
- When necessary to invoke code in other libraries from Java, on the C++ side, use JNI's RegisterNatives() function
- Data must be passed back and forth from Java to C++ mostly in the form of numbers, strings, booleans, and arrays. Easier to call Java methods from C++ than vice versa.
- Pass the C++ "this" pointer to the Java side for later use as a Java long (64-bit value).

intel®

*Other names and brands may be claimed as the property of others.

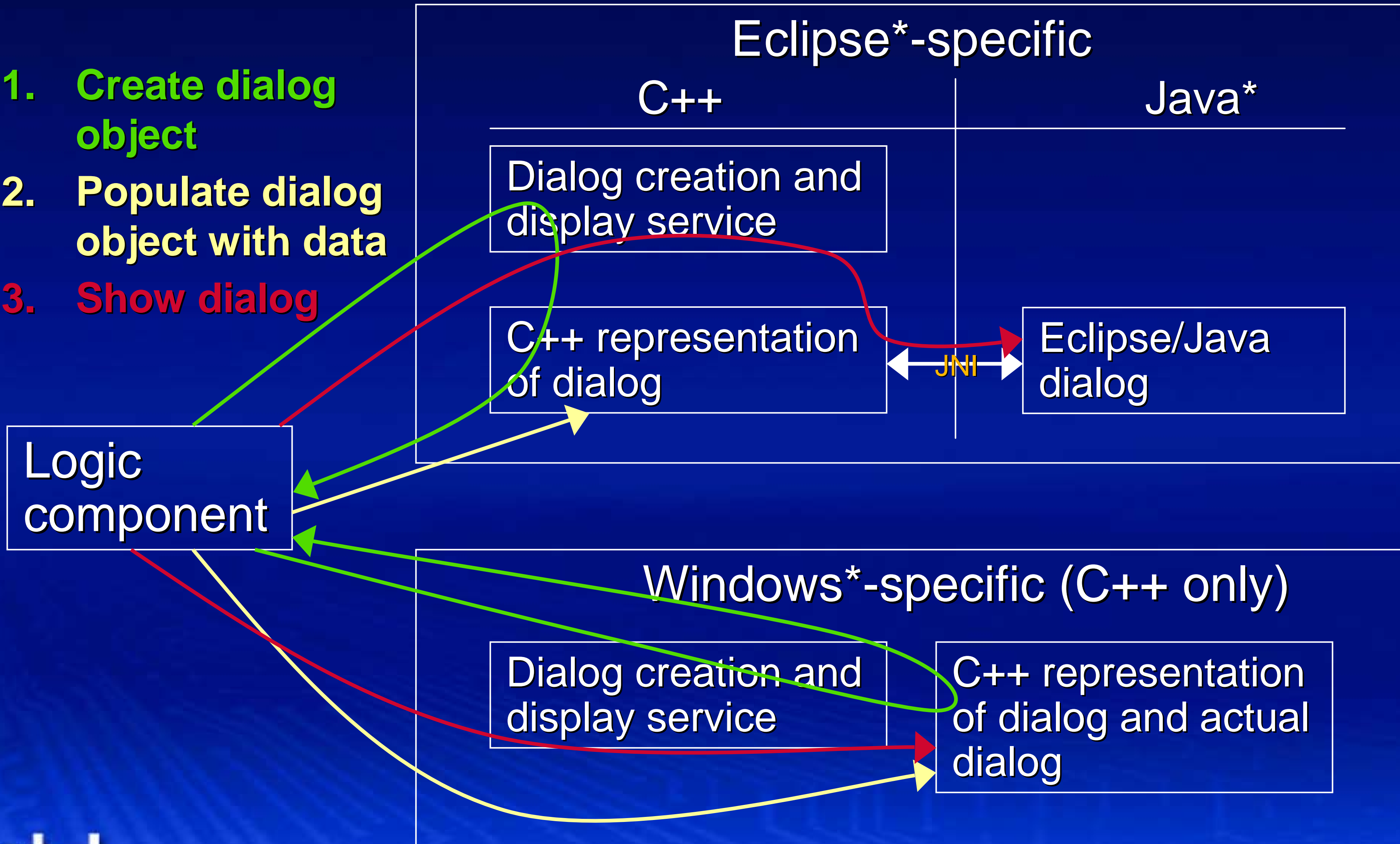
GUI Abstraction Layer (1 of 2)

- A set of standard APIs for using different types of GUI elements from the C++ side
- Different implementation for each GUI platform/environment, but the logic C++ code hooks up to it the same way regardless of the implementation
- Provides facilities for using the following different GUI elements: dialogs, wizards, property pages, preference pages, and editors

intel®

GUI Abstraction Layer (2 of 2)

1. Create dialog object
2. Populate dialog object with data
3. Show dialog



intel®

*Other names and brands may be claimed as the property of others.

Developing with Eclipse*: What is Possible

- Most capabilities for GUI development exist
- While a capability may exist, there may be little or no documentation on how to use it
- May require use of internal classes and/or interfaces
- Eclipse is open source

intel®

*Other names and brands may be claimed as the property of others.

"Exec" functions from Display class

- The "Exec" family of member functions from Eclipse's Display class allow code to be run on the user interface thread
- "Exec" functions available:
  - asyncExec: Executes a Runnable on the UI thread at the next reasonable opportunity—returns immediately
  - syncExec: Executes a Runnable on the UI thread at the next reasonable opportunity—calling thread suspended until Runnable completes
  - disposeExec: Saves Runnable for later use, which will be executed when the UI is going away
  - timerExec: Executes Runnable after input time has expired

intel®

“Exec” functions: How to Use

- asyncExec: safest of the “Exec” member functions. If code should run on the UI thread, suggested to use asyncExec() as much as possible
- syncExec: can be dangerous if called before UI is properly initialized or when UI is going away—use sparingly
- disposeExec: can be dangerous if called before UI is properly initialized or when UI is going away—best to wrap in a call to asyncExec
- timerExec: alternative: create a Timer and use asyncExec or syncExec with the Timer

intel®

Wrapping disposeExec example

```
Display.getDefault().asyncExec(new Runnable() {
    public void run()
    {
        Display.getDefault().disposeExec(new Runnable() {
            public void run()
            {
                // do something here
            }
        });
    }
});
```

intel®

Rules for Using "Exec" Functions

1. Use Display.asyncExec() as much as possible.
2. Do not use Display.syncExec() in code that may be called before the UI is initialized or when the UI is going away.
3. To use Display.disposeExec() before the UI is initialized, wrap it in a call to Display.asyncExec().
4. Use Display.timerExec() sparingly.

intel®

Using the Eclipse* Console View (1 of 2)

- A console or output view provides a way to inform users of messages without interrupting the user's execution flow, as opposed to message boxes.
- Eclipse provides a Console view; however, part of debug aspect of Eclipse, no documented API, may not be in perspective

intel®

*Other names and brands may be claimed as the property of others.

Using the Eclipse* Console View (2 of 2)

1. Find the Console view using IWorkbenchPage::findView with input of "org.eclipse.debug.ui.ConsoleView".
2. Cast to IDebugView (from org.eclipse.debug.ui) and get viewer using IDebugView::getViewer().
3. Cast viewer to TextViewer and use TextViewer::getDocument() to get IDocument.
4. Use IDocument methods to change text.

Note: Always use instanceof operator to make sure objects implement various interfaces.

intel®

*Other names and brands may be claimed as the property of others.

Proxy actions and controlling action state

- Proxy action objects are generated for each action. How to use:
  - Declare an action delegate class for the action and implement IActionDelegate2
  - The proxy action object is passed in to IActionDelegate2::init as an IAction. Use IAction methods to control the state of the action.

intel®

Cross-platform Eclipse* projects

- By default, projects created for Eclipse only work on the operating system on which they are created
- They are also specific to a particular version of Eclipse
- Example from .classpath file:

```
<classpathentry kind="var“
    path="ECLIPSE_HOME/plugins/org.eclipse.ui_2.1.0/ui.jar"
    sourcepath="ORG_ECLIPSE_PLATFORM_SOURCE_SRC/org.eclipse.ui_
    2.1.0/uisrc.zip"/>
<classpathentry kind="var“
    path="ECLIPSE_HOME/plugins/org.eclipse.ui.win32_2.1.0/workbenchwin
    32.jar"
    sourcepath="ORG_ECLIPSE_PLATFORM_WIN32_SOURCE_SRC/org.ecli
    pse.ui.win32_2.1.0/workbenchwin32src.zip"/>
```

- How to solve? Use classpath containers for dependent plug-ins and then update the classpath. Feature available since Eclipse 2.1.

intel®

*Other names and brands may be claimed as the property of others.

Ant build scripts

- Ant build scripts created by Eclipse* are generally not suitable for anything but the system on which they were created. For example:

classpath="../../../eclipse/plugins/org.eclipse.ui.win32_2.1.0/workbench.jar;../../../eclipse/plugins/org.eclipse.ui.win32_2.1.0/workbenchwin32.jar"

- How to solve? Requires modifications to build.xml
  - Use environment variable for root Eclipse directory
  - Use <path>, <fileset>, and <patternset> tags appropriately to specify dependent plug-ins in a fashion that does not depend on the OS or version of Eclipse
  - Construct appropriate classpath based on patternset

intel®

*Other names and brands may be claimed as the property of others.

Example ant build script

```
<path id="external.jars">
    <fileset dir="${env.ECLIPSE_HOME}\plugins">
            <patternset id="external.pattern">
                    <include name="org.eclipse.swt.*/**/*.jar" />
                    <include name="org.eclipse.ui_*/*.jar" />
                    <include name="org.eclipse.ui.workbench_*/*.jar" />
            </patternset>
    </fileset>
</path>

<path id="classpath">
    <pathelement path="${classpath}"/>
      <fileset dir="${env.ECLIPSE_HOME}\plugins">
            <patternset refid="external.pattern"/>
      </fileset>
</path>
.........................................................
<javac destdir="${temp.folder}/plugin.jar.bin"
    bootclasspath="${bootclasspath}">
            <src path="src/"/>
            <classpath refid="classpath"/>
</javac>
```

intel®

Eclipse* Development: Other Advantages

- Externalizing strings (for localization).
  - But does not handle XML files—have to do them by hand
- *Many* handy features for Java* and Eclipse development
  - Many features may not be immediately apparent
- "Fast track to an integrated tool suite". What does this mean? Don't have to re-invent the wheel:
  - Concept of a plug-in and pluggable features
  - Numerous UI services
  - Can focus on providing features, not designing and implementing infrastructure

intel®

*Other names and brands may be claimed as the property of others.

Conclusion

- Overall experience with Eclipse*: Positive
- Designing and implementing a GUI for the VTune™ Performance Analyzer appears to be a relatively complex task when it comes to Eclipse development—but most tasks so far have been accomplished with Eclipse.

intel®

*Other names and brands may be claimed as the property of others.