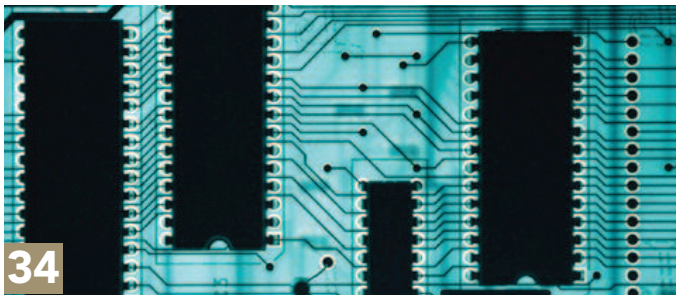
Maximizing Your Java Application Development

an internet.com Developer eBook

contents

Maximizing Your Java Application Development



While Java helps in fulfilling the promise of "write once, use anywhere", there are practical concerns developers need to address in developing their code, whether its porting from another language, working in the best IDE or optimizing for today's multi-core computing environments. This eBook will help you in understanding these issues and how you can get the most out of your Java code.

Onward and Upward: Porting Apps to Higher JDK Versions

Porting an existing Java-based application to a new JDK version is not as easy as many assume. Learn a comprehensive, systematic approach that can ensure a smooth process

by Rahul Kumar Gupta

The IT industry is synonymous with change. Every day sees some new software version or specification released, which necessitates constant upgrades. Programming professionals often must upgrade business applications to the new versions of the software upon which they are built. To accommodate these rapidly changing business requirements, Sun Microsystems releases a JDK version with some new capabilities, enhancements, and improvements nearly every year.

This article describes the process of porting an existing Java-based application to a new JDK version and prescribes a porting process that ensures the functionality of the ported application will remain unchanged (see Figure 1).

Porting is the process of making software that was written for one operating environment work in another operating environment that offers new value-added features and improved performance. Porting requires changing the programming details, which can be done at the binary (application) level or the source code level. The target configuration may include a new operating system, compiler, database, and/or other third-party software that will be integrated with the base product.

Figure 1. Porting Process Diagram: Functionality of both System A and System A' are exactly the same.

Before making the decision to port, one must determine the why, what, and how of the task. Answering the following questions in the given order will help:

1. Why migrate the existing application and/or product?

2. What in the existing application and/or product has to be migrated?

3. How do I migrate the application and/or product?

Why Port in the First Place?

What are the reasons or external events that necessitate the porting process? One good reason for porting is to take advantage of the new features and improved functionality of a newer JDK, but you still must determine whether that's a real business requirement. If none of your business applications require the new fea-

tures the newer JDK version introduces, then spending the time and money to port your existing application may not be a good idea. On the other hand, if you are in the business of developing software products, then it is mandatory to constantly upgrade and support the latest functionality on the market in order to give customers value for their money.

The following are some of the scenarios in which one should port an application or product:

1. When an existing application/product stops running on the targeted JDK [The changes introduced in the JDK, which are mandatory for implementation, can cause this (e.g., changes in the existing interfaces, changes in exception handling, introduction of new keywords such as assert in JDK 1.4, etc.)]

2. When you want to improve the performance of your existing application/product with the features introduced in the targeted JDK

3. When you want to remove the application's dependency on third-party products/APIs, which are now introduced as features of the targeted JDK (e.g., using of logging API instead of Log4j)

4. When a new feature replaces an old one and the new feature will be used in all forthcoming releases

5. When you want to use the new JDK compiler, and you want the program to compile cleanly

6. When you are in the business of product development or producing APIs, then upgrading is a business need for client requirements as well as for competitive edge

You may be thinking: if Java truly delivers "Write Once, Run Anywhere" (WORA) functionality, porting applications between JDKs should be so trivial as to make this article moot. True, but WORA is not always the reality. Sure, when you move from a Windows to a Unix platform or vice-versa, your JDK version remains the same. However, you have to keep some key points in mind while writing the code, including:

1. Naming conventions – Unix is case sensitive.

2. Use of appropriate separators (File.pathSeparator and File.separator) – Unix uses '/' and ':' as file and path separators, while Windows uses "\" and ';', respectively.

3. AWT GUI components – They need special attention. Nowadays, it is advised to use swings.

4. Threading model – In this arena, Java's WORA falls flat on its face. Making threading truly platform independent is a nightmare for programmers. For example, programmers are advised to break up thread-processing code written in the run method into smaller chunks and, if necessary, to call yield() at the end of a loop iteration. In a nutshell, they must keep the pre-emptive and co-operative models in mind while writing this code (Read this article for more information: An Introduction to Java Thread Programming)

5. Internationalization – Character sets are different in Unix and Windows.

6. Proper path usage – Avoid using absolute paths, try to use relative paths instead. If that's not possible, try to fetch from properties files.

So, with all these factors to consider, you very well may run into problems when you try to run or compile your existing application on a newer JDK version.

What Exactly Do I Need to Port?

Once you decide to go ahead with porting, your next step is to finalize what all needs to be migrated and determine their priorities. In order to ensure a smooth porting process, you must decide this and finalize your approach during the early stages of the project.

To help sort out your priorities, JDK changes can be broadly classified into four categories:

- Incompatibilities
- Improvements
- Deprecated API
- New features in JDK

These changes can be introduced in Sun APIs as well as JDK APIs. Ideally, you should not use sun.* packages because Sun intends them only for their own usage. If you are using or extending a sun.* API in your application, be ready to port the changes or incompatibles introduced in it.

Incompatibilities

Sun technically classifies the compatibility between two JDK release versions as either binary compatibilities or source compatibilities. Binary compatibilities exist when the compiled code can run with the other version of the JDK. Source compatibility means the source code can comply and run with the other version of the JDK.

Binary compatibility has two types:

- Upward – when compiled code can run with a higher JDK version
- Downward – when compiled code can run with a lower JDK version

Similarly, source compatibility also has two types:

- Upward – when source code can comply and run with a higher version the JDK
- Downward – when source code can comply and run with a lower version of the JDK

In general, maintenance releases (e.g., JDK 1.4.1 and 1.4.2) support both upward and downward compatibility at the binary and source levels. The functional releases (e.g., JDK 1.3 and 1.4) support upward compatibility at both the binary and source levels, except for Sun's stated incompatibilities. However, they do not guarantee downward compatibility at either level.

Removing the incompatibilities in the targeted JDK release is the top priority when porting your application, and it qualifies as the only MUST DO activity. Some of the incompatibilities you'll find are changes in the existing interfaces, constants, exception handling, introduction of keywords (e.g., JDK 1.4 introduced assert), and removal of some methods and constants (mainly in sun.* packages). In most cases, you can compile and run code written in JDK 1.1 with JDK 1.2, 1.3, 1.4, and 1.5, as long as you don't use any statements/APIs listed as part of the incompatibilities.

For more details about incompatibilities visit java.sun.com. The following are some other helpful links:

- Incompatibilities in J2SE 1.4.1 (since 1.4.0)
- Incompatibilities in J2SE 1.4.0 (since 1.3)
- Incompatibilities in J2SE 1.3 (since 1.2)
- Incompatibilities in J2SE 1.2 (since 1.1)

Improvements

JDK improvements could be lumped in the new features category, but for a clear distinction and a better understanding of tasks, it's better to view them as a separate task. Although these activities aren't mandatory for making your application portable, from the performance, optimization, and best practices points of view, all the changes in this category are required. Before performing all the improvement changes, however, you must weigh the trade-off between the time and effort they require and the improvement you expect to get out of them.

A few of the suggested improvements for different JDK versions are:

- With JDK 1.1, use GregorianCalendar instead of java.util.Date for wider acceptability.
- With JDK 1.2, use Arraylist instead of Vectors, if you are not required to synchronize.
- With JDK 1.3, use the Timer class for scheduling future execution in a background thread instead of your own implementation.
- With JDK 1.4, the Preference API is better than the properties file for managing user preferences and configuration data.

Deprecated API

Deprecation APIs are those that have been restructured and modified with new classes and methods that provide similar functionality. In general, whenever an API is deprecated, an alternative implementation is provided and information about it is offered in the API's javadoc. Sun warns its users to withdraw support for deprecated APIs in future releases. Whenever possible, you should modify your application to remove references to deprecated methods/classes and to use the new alternatives. Deprecated APIs support methods and classes only for backward compatibility, and your Java compiler will generate a warning whenever you use them.

Marking something as deprecated is only one aspect of documentation and is not part of the OO paradigm. Deprecation also is not inherited, so you can still override a deprecated method without treating the subclass methods as deprecated methods. For example, say Class X has a deprecated method called getStringValue(), and Class Y extends Class X, overriding the getStringValue () method. While compiling test client TestX, which creates an instance of Class X and calls getStringValue(), the compiler generates a warning. However, while compiling test client TestY, which creates an instance of Class Y and calls getStringValue (), it doesn't generate a warning.

You could treat a deprecated API as part of source compatibilities, but you should consider it a separate activity. Currently, you don't have to eliminate the usage of deprecated APIs in order to port applications. In fact, the deprecated APIs from JDK 1.1 are still available in JDK 1.5 (J2SE 5). However, Sun still may withdraw their support in future releases.

Click here for a complete list of deprecated APIs still available in JDK 1.5.

New Features in JDK

Every major JDK version release provides new features in the form of new APIs and/or extensions to existing APIs. The following are some of the major features introduced in respective JDK versions:

1. JDK 1.1 introduced inner classes.

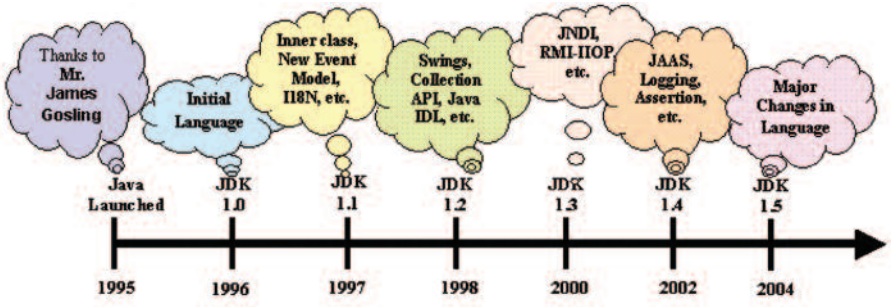
2. JDK 1.2 introduced swings, JDBC 2.0, and changes in Java security.

3. JDK 1.3 introduced JNDI, RMI-IIOP, Java Sound, enhanced the Collection framework, and completed swings.

4. 1.4 introduced the JAAS Preferance API, the Logging API, JDBC 3.0, the assertion facility, and Regular expression.

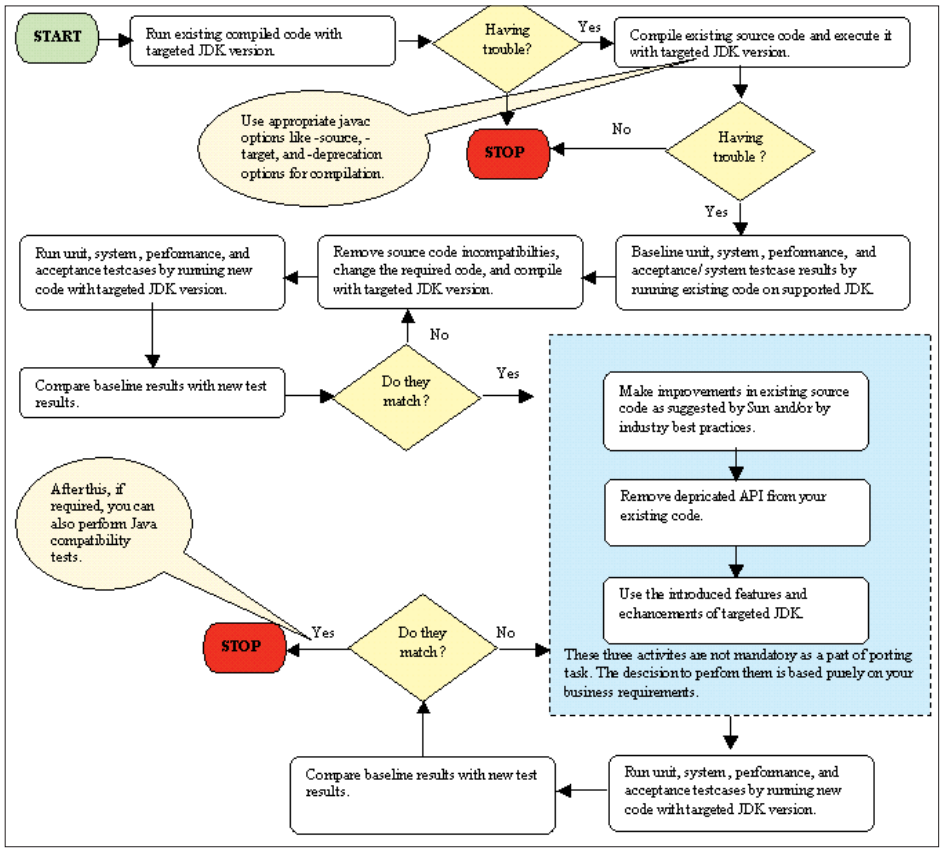
5. 1.5 introduces major changes at the language level including autoboxing/unboxing, enhanced for loop, static import, typesafe enums, and varargs. For a list of new features in your targeted JDK, refer to its javadoc. Effectively using the targeted JDK's new features, as well as its enhanced existing features, in your existing application is part of the porting process. Sometimes, using newly added fea-

Figure 2. JDK Releases and the Major Feature They Introduced



tures ends your dependency on third-party software/APIs (e.g., companies that previously used log4j for logging can now use the logging facility in JDK 1.4 instead). However, if you use new features just to enhance your existing application, then it's an enhancement activity rather than a porting activity. Enhancement activities should not be included as part of the porting process because they may lead to some regressions in existing functionality, and then determining whether these problems are because of the porting activity or because of the enhancement becomes very difficult. Still, whether or not to go for the new features is purely dependent on your business requirements. It is not a mandatory part of the porting task.

Figure 3. Porting Process Flow Diagram



How Do I Port My Application?

Once you have identified the porting tasks that need to be performed, as well as their priorities, make an action plan to execute the porting process. Testing plays a very important role in this execution. Create base line results or use existing base line results for unit, system, performance, and acceptance testing of your current system for each platform it supports. Figure 3 shows a porting process flow diagram.

Before starting the porting process, take the following steps:

1. Prepare a list of the following changes:
   - Incompatibilities introduced in Java API and Sun API for targeted JDK version
   - Deprecated APIs
   - Suggested improvements
   - Newly added features

Try to estimate the number of occurrences of each identified change in the entire application source code (i.e., what is the approximate number of places you have to make changes corresponding to each JDK change introduced). Also estimate the time required for each change. Table 1 offers a sample template for capturing these details.

Table 1. Template for Capturing Change Details

S. No.	Priority (High, medium low)	Changes inttroduced or to be done	Complexity (high, medium low)	Sub Category (JDK feature) -- Swings/ AWT/Exception IO, etc.	Description (API)	Changes Done (Source file names, method name, or method name or line no.)	Remarks
1.							
2.							

2. Prepare baseline results.
Consider a scenario in which a CRM product supports the following:

- ATG and BEA WebLogic application servers
- Windows, Linux, and Solaris operating systems (OSs)
- JDK 1.3

Table 2. Template for Capturing Test Results

S. No.	Modules	Old System						Ported System					
		ATG			WebLogic			ATG			WebLogic		
		Pass	Fail	Skip	Pass	Fail	Skip	Pass	Fail	Skip	Pass	Fail	Skip
Unit TestCase													
1													
2													
System TestCases													
1													
2													
Performance TestCases													
1													
2													
Acceptance TestCases													
1													
2													

The product needs to be ported to JDK 1.4. As discussed previously, comparing the test results on the ported platform with the original (baseline) test results is crucial. So the baseline results and actual results must be captured. Table 2 offers a template for capturing test results.

Capture the test results for each of the supported OSs (Windows, Linux, and Solaris) separately. If all the baseline results match the new test results of the ported application, then your porting process is almost complete.

Another advisable practice is performing Java compatibility tests as per your requirements [e.g., Pure Java Check and J2EE CTS (Compatibility Test Suite)]. Such compatibility ensures that implementations of Java technology meet Java specifications.

Table 3 presents some of the changes that you might have to perform while porting an application from JDK 1.3 to JDK 1.4.

Table 3. Changes When Porting an Application from JDK 1.3 to JDK 1.4

S. No	API	JDK 1.3	JDK 1.4
AWT/SWINGS			
1	javax.swing.tree.DefaultTreeModel	Sets the root to root, throwing an IllegalArgumentException if root is null	Sets the root to root; a null root implies the tree will display nothing and is legal
2	javax.swing.text.DefaultHighlighter. DefaultPainter	Non-final	Now it is final, so make the changes if you are changing/overriding this value
3	java.awt.event.MouseEvent. MOUSE_LAST	The value is 507	Now changed to 506
CORBA			
4	Helper Classes		All Helpers are now abstract public classes
5	java.util.Calender		Now Calendar writes a ZoneInfo object in its writeObject method. Similarly, it calls readObject to read this. Based on Java object serialization, when talking to an older version, it expects the object not to be there, so the stream will throw an EOFException and keep the stream position intact.
6	org.omg.CORBA.ORB		ORB.init method is changed and now requires explicit type casting:
			orbObj=(com.sun.corba.se.internal.iiop.ORB) ORB.init ((String []) null, orbProperties);
7	com.sun.corba.se.internal.CosNaming TransientNameService	Earlier it was used like TransientNameService (mOrb)	Now TransientNameService ((com.sun.corba.se.internal.POA.POAORB) mOrb)

continued

S. No	API	JDK 1.3	JDK 1.4
8	Helper classes		All Helpers are now abstract public classes.
JDBC			

9	java.sql. CallableStatement,
	java.sql. DatabaseMetaData,
	java.sql. PreparedStatement,
	java.sql. ResultSet,
	java.sql. Statement
			New methods are being added in these interfaces so as to provide at least blank implementation like this:
			public boolean execute(
			String sql, String columnNames[])
			throws java.sql.SQLException
			{
			throw new java.sql.SQLException(
			"JDBC 3.0 not implemented yet");
			}

EXCEPTION HANDLING			
10	java.awt.Toolkit java.awt.GraphicsEnvironment java.awt.Cursor java.awt.print.PrinterJob javax.swing. JOptionPane	Some methods in these classes either throw no exception or throw some other exceptions	Now, they throw Headless Exception. So you need to make the required changes in the code to catch these exceptions.
11	java.applet java.awt java.awt.dnd java.awt.print javax.print javax.swing	Constructors of some of the previous classes throw no exception or throw some other exceptions	Now, they throw a Headless Exception. So you need to make the required changes in the code to catch these exceptions.
12	java.util.ResourceBundle	Earlier, some methods like: 1.Object getObject(String) 2.String getString(String) 3. String[] getStringArray(String) throw MissingResourceException	Now, they throw no exception.
13	URLConnection.getInputStream	Throws FileNotFoundException	Now, throws IOException for all HTTP errors regardless of the file type, and throws FileNotFoundException if the response code is 404 or 410.
LANGUAGE SPECS			
14	Keyword introduced		Assert is now a keyword, so change the code anywhere you used it as a method name, variable, or class.
15	Empty statements	Works fine return ;;	Now, compiler gives error as Unreachable statement at second semicolon, so change it to
16	Unnamed namespace	Works fine import SomeClass ;	return; Now, this has to be changed to import somepacke.SomeClass; The specification is being clarified to state clearly that you cannot have a simple name in an import statement, nor can you import from the unnamed namespace.

continued

S. No	API	JDK 1.3	JDK 1.4
Sun packages (sun.*) - Only get effected when you are extending or using these packages			
17	com.sun.tools.doclets com.sun.javadoc.PackageDoc com.sun.javadoc.Doc com.sun.javadoc.DocErrorReporter com.sun.javadoc.DocErrorReporter		Some new abstract methods are added in these classes. So implement the body in leaf class.
18	com.sun.tools.doclets.standard	isGeneratedDoc() is available	It is no more available, instead it is call: Standard.configuration().isGeneratedDoc(cd));
19	com.sun.tools.doclets.standard.ClassWriter		Constructor is changed

The areas where you'll notice the majority of changes across different JDK versions are security, AWT, swings, I/O, exception handling, RMI/CORBA, and the collection API.

A Systematic Approach Covers All the Bases

The process and strategy this article describes may not be the only way to handle JDK porting activities, but from my experience with these types of project, it provides a comprehensive, systematic approach that can ensure a smooth process.

Credits: The author wishes to thank his project manager Atul Jain and his colleague Saurabh Bhatnagar for their valuable contributions as reviewers for this article.

Rahul Kumar Gupta *has more than six years' experience in the IT industry, particularly working with EAI, J2EE, design patterns, UML Java, RMI, Servlets, JSP, application servers, C, CORBA, WAP, and J2ME. He currently works with Indian IT company HCL Technologies Limited, NOIDA (India).*

This content was adapted from DevX.com's Web site.

Automate Your J2ME Application Porting with Preprocessing

Got porting nightmares? If you're considering automating the porting your J2ME applications, you may want to think about using a preprocessor. Find out why it's the only technique open-ended enough to handle porting to multiple device models.

by Bruno Delb

Because Java is theoretically portable, people assume that when you develop a Java mobile application, it should run correctly on all Java-enabled devices. Like most things theoretical, this just doesn't work in the real world. During the short life of J2ME mobile applications, many developers have expressed concerns that inter-operability problems are not going to be solved so easily by new initiatives like MIDP 2.0 or JTWI.

The reality is that J2ME may be globally portable but J2ME applications are not. This means that byte-code runs correctly on all Java handsets but the behavior of an application must be adapted for almost each handset. There are 1200 mobile devices, all of which have different capabilities, support various Java platforms-including MIDP, and support optional APIs and optional parts of APIs. Not to mention that each of these implementations has its own unique set of bugs (See the Sidebar: The Fragmentation of device Characteristics and Features).

The result of this is that, in a typical development cycle, porting and testing can consume from 40 to 80 percent of your time, depending on your level of experience and on the number of devices you need to support. Testing your ported mobile applications on real mobile phones isn't always easy. A good emulator should reproduce all the bugs of the real device, but emulators don't always exist and when they do, they're far from reliable.

Jupiterimages

Bugs are another difficulty- you either need resolve bugs in your source code or deactivate the problem features, all of which can be different depending on the firmware version. Manufacturers face difficulties when there are bugs in their virtual machines. This causes problems with the integration of the VM in the handset at the hardware level.

Mobile operators need to control the quality of the distributed midlets, because low-quality midlets affect operator service. Operators also need to transpose older, more successful midlets for newer device models and for wider availability. This is why developers have to be able to port their applications to these new devices quickly.

Do You Need Automatic Porting?

The number of devices your application is able to support is the main question – and the one that most effects your return on investment. Automating your porting has many benefits:

- It reduces the time to market and enables J2ME

prototypes.
- It automates tedious tasks.
- It allows you to focus on your application logic instead of various device specs.
- It allows you to produce optimized applications to each device.

First, you'll need take into account the specificities of each device model. Your application should use optional features whenever available. Table 1 shows some examples of what functionalities your application should support depending on the device.

In-house Porting Solutions

So, you've decided you do need to automate porting your application. Sure, you could outsource it, but if you're thinking about doing it in-house, you've basically got four methods from which to choose.

1. Build one Version per Device Series: This approach is to simply develop an application for a specific series of models, for example, the Nokia Serie40 Edition 1. The problem here is that the more APIs you support, or the more your application stresses the device, the more you fragment the series since support for advanced APIs highlights the small differences within the series. For example, two similar devices will have wildly varying performance results due to the number of images on the screen.

2. Dynamic Detection of the Handset: This option involves testing your application during execution. For example, suppose your model is a Nokia handset. Your application would detect the device model during execution and select the appropriate behavior depending on the model.

This allows you to call methods only available on specific handsets – like fullscreen mode. You have to create one class for each specific implementation (NokiaCanvas, SiemensCanvas, and StandardCanvas). The following code demonstrates:

Table 1: Examples of the specificities of device models

If the device…	…Your application should
supports sounds	play sounds
supports alpha-blending	display the menu by varying opacity
has strong .jar file size limitations	remove not mandatory images
supports fullscreen	use fullscreen
has enough heap memory and supports large .jar file sizes	use optional features like bitmap fonts
supports camera control	be able to customize games using snapshots taken with the camera
supports Bluetooth	use the SMS or HTTP connections to communicate with others users (ie. chat, etc.)
supports SMS	use it to pay to activate the current application
supports PDAP	use it to access images in an internal gallery
can initiate a phone call	make phone calls
supports running applications in background	run applications in the background

```
try {
   Class.forName("com.nokia.mid.ui.FullCanvas");
   Class myClass = Class.forName("NokiaCanvas");
   myCanvas = (ICanvas)(myClass.newInstance());
} catch (Exception exception1) {
   try {
      Class.forName("com.siemens.mp.color_game.GameCanvas");
      Class myClass = Class.forName("SiemensCanvas");
      myCanvas = (ICanvas)(myClass.newInstance());
   } catch (Exception exception2) {
      myCanvas = (ICanvas) new StandardCanvas();
   }
}
```

You basically create an interface, Icanvas, and three implementations, one for Nokia devices, one for Siemens devices, and another one for standard MIDP devices.

Then you use Class.forName in order to determine whether a proprietary API is available. If no exception is thrown, you use the NokiaCanvas. Otherwise, it means the current device doesn't support this API. In this case, you test another API (for example, Siemens). If another exception is thrown, it means you have to use the standard canvas.

Because this solution supposes the inclusion of the logic of the behavior of each the model of a device in each application, it quickly becomes impossible.

3. Using Alteratives Like AOP or Extended Java: In his DevX article, Allen Lau discussed using AOP to solve the problem of fragmentation. His idea is to concentrate the application logic in one location and to modify the code of the application by adding or removing portions of code.

This approach solves some problems, but having to adapt the structure of your application to different platforms may, in the end, force you to use another, complementary solution. This is because it can be very difficult to optimize the application to each device model – especially to support optional APIs.

Additionally, this method does not solve the problem of adapting the content (images, sounds, etc.) to each device model. You can use Java extensions to automatically transform your source code, which is an interesting approach, but it really increases work load.

4. Using a Preprocessor: Using a preprocessor, your source code will be automatically activated or deactivated depending on certain conditions.

For example, to set the full screen mode on a Nokia device, you have to extend FullCanvas, not Canvas. On a MIDP 2 device, you have to call setFullScreenMode. On a MIDP 1 device, this isn't possible, so you stay in a non-fullscreen mode.

```
//#ifdef NOKIA
   extends com.nokia.mid.ui.FullCanvas
//#else
   extends Canvas
//#endif
   {
...
```

```
//#ifndef MIDP2
    setFullScreenMode(true);
//#endif
```

A preprocessor processes this source code, then you set the directives. So, to generate the application for a Nokia device:

```
//#define NOKIA
```

The preprocessor produces:

```
  extends com.nokia.mid.ui.FullCanvas
  {
```

For a MIDP 2 device, ("//#define MIDP 2"), it produces:

```
  extends Canvas
  {
    setFullScreenMode(true);
```

This solution allows for one body of source code to be adapted to each device model. You need only develop to the reference source code, including the directives. All other modifications made to the processed files will be lost after the next preprocessing.

Though this solution relies on the old concept of preprocessing, this is the only technique open-ended enough to solve all the problems you'll encounter trying to port to multiple device models.

What's Involved in Porting with a Preprocessor?

The basic principle is to keep only one version of your source code, which is then preprocessed to generate code adapted to each device model.

Here are some things to keep in mind:

- You'll need a thorough knowledge of the specific behavior of each device model.
- You'll need to know the Java features supported by each device model.
- You'll need to know which operations are more or less covered by pre-processing (things like deployment).
- Remember that pre-processing does not convert resources (things like images or sounds).
- You need to completely automate your solution – a parameterized solution is not sufficient.
- You'll need to invest in the development of an automated compilation and packaging process.

You'll need to use conversion tools to adapt resources like images and sounds to the capabilities of each device model. The images will have to be optimized; for instance, you'll need to remove optional information in the headers of .png images, group small images in big images, reduce the number of colors for each image, preload resources, etc.

You may have to create a series of Java devices with the same characteristics, features, and behavior. Then, you can produce one application for this series, not for each device model. The features of this series will depend essentially on the features you want to use in your applications. The more optional features you use, the more fragmentation you'll be dealing with.

You'll need to update the application in order to take into account unexpected resources. For example, if the device supports big .jar files and has a high heap memory, you can store a background image for your app. Otherwise, the image will not be included in your .jar file and you will need to draw the image simply. In this case,

the .jar size will decrease and the heap memory will be less consumed.

Remember, screen size is an important issue when it comes to mobile programming, so don't hesitate to use all the available techniques to reduce image size (like dynamic transformation of images or transformation before packaging).

Another good practice is to manage the interactions with the system. For example, during an incoming call, stop sounds and pause your midlet during the call.

A Concrete Example

Suppose you want to port the game MyGame on the following devices: MIDP 1, MIDP 2, MIDP 1 NokiaUI, and MIDP 1 Motorola.

Here's a step-by-step overview of the process:
1. Download and install Ant and Antenna.

2. Create a specific class for the sound. This class will include the specific code to play sounds for each specific device.

3. Create a series of Antenna XML files for each device model (ie MIDP 1, MIDP 2, MIDP 1, NokiaUI, and MIDP 1 Motorola). These files will:
- Preprocess the source files, retaining only the lines concerning the current device.
- Build the project by compiling the source files and pre-verifing the classes.
- Run the ported midlet in the emulator for testing.

Seems easy, right?

Creating the SoundManager Class

The SoundManager class contains the instructions to play a basic sound and to set the backlight. It supports the following generic devices:

- MIDP 1: This device doesn't support sounds and backlight, so the methods will be empty.
- MIDP 2: This device supports sounds and backlight, so the lines of code between //#ifdef MIDP2 and //#endif will be selected.
- MIDP 1 NokiaUI: This device supports sounds and backlight, so the lines of the code between //#ifdef NOKIAUI and //#endif will be selected.
- MIDP 1 Motorola: This device supports only backlight, so the method to play a sound will be empty.

Listing 1 shows the code.

Listing 1.

The *SoundManager* class.

```
//#ifdef MIDP2
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.media.control.ToneControl;
//#endif
//#ifdef NOKIAUI
```

```
import com.nokia.mid.sound.*;
import com.nokia.mid.ui.*;
//#endif
//#ifdef MOTOROLA
import com.motorola.multimedia.*;
//#endif
import javax.microedition.lcdui.*;
import java.io.*;

public class SoundManager {

  Display display;

  public SoundManager(Display display) {
    this.display = display;
  }

  public void doLight() {
//#ifdef MIDP2
    display.flashBacklight (duration);
//#endif
//#ifdef NOKIAUI
    try {
      DeviceControl.setLights (0,100);
    } catch (Exception exception) {
    }
//#endif
//#ifdef MOTOROLA
    try {
      Lighting.backlightOn();
    } catch (Exception exception) {
    }
//#endif
  }

  public void doSound() {
//#ifdef MIDP2
    try {
      InputStream is = getClass().getResourceAsStream("music.mid");
      Player audioPlayer = Manager.createPlayer(is, "audio/midi");
      audioPlayer.start();
    } catch (IOException ioe) {
    }
//#endif
//#ifdef NOKIAUI
    Sound sound = new Sound (1000, 100);
    sound.play(1);
//#endif
  }
}
```

Creating the BUILD.XML File

In the BUILD.XML file, you will select the appropriate directories. It's a manual operation, so, it's best to create the XML files for each device and keep them.

In the case of the MIDP 2 profile, the preprocessing of the file SoundManager.java will produce this output:

```
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.media.control.ToneControl;
import javax.microedition.lcdui.*;
import java.io.*;

public class SoundManager {

  Display display;

  public SoundManager(Display display) {
    this.display = display;
  }

  public void doLight() {
    display.flashBacklight (duration);
  }

  public void doSound() {
    try {
      InputStream is = getClass().getResourceAsStream("music.mid");
      Player audioPlayer = Manager.createPlayer(is, "audio/midi");
      audioPlayer.start();
    } catch (IOException ioe) {
    }
  }
}
```

For the MIDP 1 profile, it produces:

```
import javax.microedition.lcdui.*;
import java.io.*;

public class SoundManager {

  Display display;

  public SoundManager(Display display) {
    this.display = display;
  }

  public void doLight() {
  }

  public void doSound() {
  }
}
```

You select the profile of the device in this line:

```
<wtkpreprocess srcdir="src" destdir="output\src" symbols="MIDP2" verbose="false"/>
```

Antenna takes the content of the attribute symbols (MIDP 2 in this example) and inserts //#define MIDP2 at the beginning of each file, like this:

```
//#define MIDP2
//#ifdef MIDP2
import javax.microedition.media.*;
import javax.microedition.media.control.*;
import javax.microedition.media.control.ToneControl;
//#endif
//#ifdef NOKIAUI
import com.nokia.mid.sound.*;
import com.nokia.mid.ui.*;
//#endif
//#ifdef MOTOROLA
import com.motorola.multimedia.*;
//#endif
import javax.microedition.lcdui.*;
import java.io.*;
```

Building the Preprocessed Source Code

Next, build the preprocessed source code, which means compiling the two files you've created.

The name of the .jad file and its content are specified in the tag:

```
<wtkjad jadfile="output\bin\${midlet.name}.jad"
  jarfile="output\bin\${midlet.name}.jar"
  name="${midlet.name}"
  vendor="You"
  version="1.0"
  target="">
  <midlet name="${midlet.name}" icon="/icon.png" class="game.${midlet.name}"/>
  <attribute name="MIDlet-Icon" value="/icon.png"/>
</wtkjad>
```

It's advisable to always use the same name for the icon (for instance, icon.png).

To package the midlet, create the .jar file. Select the resources to be integrated in your .jad file. Here are some simple guidelines for this process:

- Separate graphics for different types of screens (big screens, medium screens, small screens, etc.).
- Separate the icons of each size in a sub-directory (res_icon32x32, res_icon16x16, etc.).
- Separate the resources for sounds (res_midifiles, res_ottfiles, etc.).

In Table 3, the left column describes the code in the right column.

Table 3. Building the Preprocessed Code

	<?xml version="1.0"?>
Name the project.	<project name="MyMidletProject" default="build" basedir=".">
Specify where to find WTK.	<property name="wtk.home" value="c:\WTK23\"/>
Specify the midlet name.	<property name="midlet.name" value="MyFirstMidlet"/>
	<property name="midlet.home" value="${wtk.home}/apps/ ${midlet.name}"/>
Specify the standards APIs you use (WMA, MMAPI, PDAP, 3D, Bluetooth, Web services).	<property name="wtk.wma.enabled" value="false"/> <property name="wtk.mmapi.enabled" value="true"/> <property name="wtk.optionalpda.enabled" value="false"/> <property name="wtk.java3d.enabled" value="false"/> <property name="wtk.bluetooth.enabled" value="false"/> <property name="wtk.j2mews.enabled" value="false"/>
Specify the CLDC version.	<property name="wtk.cldc.version" value="1.0"/>
Specify the MIDP version (MIDP 1 or MIDP 2): - <property name="wtk.midp.version" value="1.0"/> - <property name="wtk.midp.version" value="2.0"/>	**<property name="wtk.midp.version" :value="2.0"/>**
Specify the proprietary APIs (Nokia Serie40, Nokia Serie60, …). - <property name="wtk.midpapi" value="c:\libs\nokia_s40\classes.zip"/> - <property name="wtk.midpapi" value="c:\libs\nokia_s60v2\j2me-debug.zip"/>	
	<taskdef resource="antenna.properties"/>
Clean the classes directory.	<target name="clean"> <delete failonerror="false" dir="classes"/> <delete failonerror="false"> <fileset dir="."> <exclude name="build.xml"/> </fileset> </delete> </target>
	<target name="build">
Clean the output directories: output\bin, output\classes, output\src	<delete dir="output\bin" verbose="false"/> <delete dir="output\classes" verbose="false"/> <delete dir="output\src" verbose="false"/> <mkdir dir="output\bin"/> <mkdir dir="output\classes"/> <mkdir dir="output\src"/>
Launch the pre-processor process	<wtkpreprocess srcdir="src" destdir="output\src" symbols="MIDP2" verbose="false"/>
Launch the compilation process	<wtkbuild srcdir="output\src" destdir="output\classes"/>
Launch the packaging process	<wtkjad jadfile="output\bin\${midlet.name}.jad" jarfile="output\bin\${midlet.name}.jar" name="${midlet.name}"

Table 3. Building the Preprocessed Code *continued*

	vendor="You" version="1.0" target=""><midlet name="${midlet.name}" icon="/icon.png" class="game.${midlet.name}"/> <attribute name="MIDlet-Icon" value="/icon.png"/> </wtkjad>
	<wtkpackage jarfile="output\bin\${midlet.name}.jar" jadfile="output\bin\${midlet.name}.jad" obfuscate="false" preverify="false"> <fileset dir="output\classes"/>
Include the resources (for the big screens or for the small screens). - <fileset dir="res_bigscreen"/> - <fileset dir="res_smallscreen"/>	<fileset dir="res_bigscreen"/>
Include the icon (format 32x32 or 16x16) : - <fileset dir="res_icon_32x32"/> - <fileset dir="res_icon_16x16"/>	<fileset dir="res_icon_16x16"/>
Include the sounds files (midi files or ott files) : - <fileset dir="res_midifiles"/> - <fileset dir="res_ottfiles"/>	<fileset dir="res_midifiles"/>
	<preserve class="game.${midlet.name}"/> </wtkpackage>
	</target> </project>

Creating the RUN.XML File

To run the emulator, you have to create a XML file. The emulators have to be installed in the directory wtklib\devices of the WTK. You have to use the device name displayed in WTK (the directory name for the device) and specify it in the following line in the attribute device:

<wtkrun jadfile="output\bin\${midlet.name}.jad" device="DefaultColorPhone"/>

Table 4's left column explains the code for the RUN.XML file, shown in the right column.

This example supports basic sounds and backlight for some devices. But in reality, you'd also have to support other features like keyboard constants, fullscreen support, screen refresh loop, regulation of frame rate, and image and sound formats.

Addressing the Complete Chain of Production

Remember that deployment and testing are also very important. Because each small, manual action has to be repeated for each device model, mobile development can quickly become very tedious. Imagine renaming all the .jad files, the .jar files, and modifying the content of each .jad file, which includes the name of the .jar file. Or imagine that your upload to the test WAP server for test purposes is manual, forcing you to use your FTP client for 300 or 400 devices!

Table 4. The left column explains the code for the RUN.XML file, shown in the right column.

	<?xml version="1.0"?>
	<project name="MyMidletProject" default="build" basedir=".">
	<property name="wtk.home" value="c:\WTK23\"/>
	<property name="midlet.name" value="MyFirstMidlet"/>
	<property name="midlet.home" value="${wtk.home}/apps/${midlet.name}"/>
	<property name="wtk.wma.enabled" value="false"/> <property name="wtk.mmapi.enabled" value="true"/> <property name="wtk.optionalpda.enabled" value="false"/> <property name="wtk.java3d.enabled" value="false"/> <property name="wtk.bluetooth.enabled" value="false"/> <property name="wtk.j2mews.enabled" value="false"/>
	<property name="wtk.cldc.version" value="1.0"/>
	<property name="wtk.midp.version" value="1.0"/>
	<taskdef resource="antenna.properties"/>
	<target name="build">
Run the emulator (WTK, Nokia Serie40 or Nokia Serie60) : - For WTK : <wtkrun jadfile="output\bin\${midlet.name}.jad" device="DefaultColorPhone"/> - For Nokia Serie40 Edition 1 : <wtkrun jadfile="output\bin\${midlet.name}.jad" device="Nokia_7210_MIDP_SDK_v1_0"/> - For Nokia Serie60 <wtkrun jadfile="output\bin\${midlet.name}.jad" device="Series_60_MIDP_SDK_for_Symbian_OS_v_1_2_1"/>	<wtkrunjadfile="output\bin\${midlet.name}.jad " device="DefaultColorPhone"/>
	</target> </project>

As strategic as porting is, it's always good to take into account all the optional APIs (like Bluetooth, 3D, file connection, SMS, MMS, etc.) and the optional parts of APIs (like camera support, audio recording, .jpeg, etc.). You shouldn't continue to produce the same application on all the devices while some devices can propose more powerful features.

Sidebar: *The Fragmentation of device Characteristics and Features*

DEVICE CHARACTERISTICS	DEVICE FEATURES	MOBILE OPERATOR CHARACTERISTICS
• Screen size • Maximum .jar size • Heap memory • Maximum RMS size	• Bugs in the implementation • Platforms: MIDP 1/MIDP 2/ DoJa 1.5/DoJa 2.5/ Applets • Proprietary APIs: VSCL, NokiaUI, LG API • optional APIs: JSR 184, JSR 82, JSR 205	• Gateway characteristics • opened ports • Device-side restrictions (on APIs access, open/closed access to outside the network)

Device Capabilities

JAVA VERSIONS	PROPRIETARY APIs	STANDARDS APIs
Some platforms are geographic region-specific	*Due to the lack of MIDP 1*	*Some are optional or have optional parts*
• MIDP 1, MIDP 2 (international) • DoJa 1.5, DoJa 2.5 (for Europe) • Brew J2ME (bridge), iDEN (for the US) • DoJa 1.0, DoJa 2.0, DoJa 3.0, DoJa 4.0 (for Japan)	• VSCL • NokiaUI • Motorola • Siemens • LG • Samsung • Sprint • Etc …	• Optional APIs: WMA 1 (for SMS), WMA 2 (for MMS), BTAPI (for Bluetooth), LBS (for geolocalization), 3D, JavaCard/ J2ME bridge (with JSR 177), Web services • Optional parts of APIs: MMAPI (with optional camera control, audio control, and video), WMA1 (with CBS support), BTAPI (with the various protocols)

Java Platforms

MIDP 1	MIDP 2	MIDP 3
Lacks some important features	*New features*	*Specifications is in progress, some examples of requests*
• No fullscreen support • No image transparency support (even if generally implemented by manufacturers) • No sound support • No user components at the low-level UI • No access to the environment (camera, address book)	• Game API • Applications signing • Image transparency support • Improvement in handling key presses • Sound support • OTA (Over The Air) mandatory • Push registry • Enhancement graphics (like alpha blending)	• Auto-launching midlets • Inter-midlet communication • Improved user interface • Secure local storage • Provisioning improvement • Localization & internationalization improvement

MIDP Platforms

OPTIONAL APIs	OPTIONAL PARTS OF OPTIONAL APIs
• WMA 1 (for SMS) • WMA 2 (for MMS) • BTAPI (for Bluetooth) • LBS (for geolocalization) • 3D • JavaCard/J2ME bridge (with JSR 177) • Web services	• MMAPI (with optional camera control, audio control, and video) • WMA1 (with CBS support) • BTAPI (with the various protocols)

While porting is a fundamental problem, it's not a deal breaker for producing optimized and rich mobile applications. Ultimately, a completely interoperable mobile application is not possible due to mobile device structure. The trend is to embed more and more increasingly complex software, which means that implementations will always have problems.

***Bruno Delb**, the author of the first French book about J2ME, is the founder of Net Innovations and Unified Mobiles. Unified Mobiles has created the concept of unified mobile applications development, based on UMAK (Unified Mobile Application frameworK). UMAK is a complete framework to facilitate and accelerate the development of multiplatform applications (J2ME, DoJa, and Web applets). It's based on a very detailed knowledge base of devices, a Java testing suite, a productivity tools suite, and on a configuration engine to take into account each feature of each device model.*

This content was adapted from DevX.com's Web site.

Eclipse, NetBeans, and IntelliJ: Assessing the Survivors of the Java IDE Wars

Get a comprehensive comparison of the latest versions of the major IDEs in the Java development space: NetBeans, Eclipse/MyEclipse, and IntelliJ IDEA. Find out how well each performs in four common areas of development: Swing, JSP/Struts, JavaServer Faces, and J2EE/EJB 3.0.

by Jacek Furmankiewicz

Ever since Eclipse burst out on the Java scene a few years ago, things have gotten very interesting for Java developers. With SWT and Swing toolkits both having their own strengths and weaknesses but none having any noticeable lead over the others in terms of pure performance or look-and-feel, Eclipse focused the competition among Java IDEs where it belongs: features, ease of use, and productivity. This article explores what the past few years of fierce competition within the Java IDE space (and of course indirectly with Microsoft Visual Studio.NET) have delivered.

Jupiterimages

It reviews the three major Java IDEs – NetBeans, IntelliJ IDEA, and Eclipse – from the viewpoint of basic, common features (installation, performance, editor, etc.), but it really focuses more on their strengths in four common areas of development: Swing, JSP/Struts, JavaServer Faces (JSF), and J2EE/EJB 3.0. Wherever possible, it also evaluates JPA (Java Persistence API) support, instead of hard-coded JDBC queries or particular libraries (such as Hibernate or Oracle TopLink).

Out of the three IDEs, Eclipse is the only one that exists in multiple versions/distributions, starting from the base distribution to pre-packaged ones with extra open-source plugins (such as EasyEclipse) and open-source/commercial hybrids such as Genuitec's MyEclipse. In order to provide a fairly realistic review of what Eclipse is capable of, I focused on the base distribution (including default Eclipse sub-projects such as the Visual Editor and Web Tools Project). Wherever I felt it was lacking, I also considered what MyEclipse offers as a commercial alternative. Frankly, at a subscription price of $49/year, I'd be hard pressed to find any commercial IDE with the functionality that MyEclipse provides.

Up first, though, is NetBeans 5.5.

Author's Note: As an employee of Compuware Canada, I use the Eclipse-based, Model-Driven Java development tool Compuware OptimalJ. However, I have made every effort to ensure a fair review for each IDE, with no preferences to Eclipse.

NetBeans 5.5

Vendor:	Sun Microsystems
Website:	www.netbeans.org
Price:	Free/Open source
Distribution:	Base + Enterprise Pack + Visual Web Pack

NetBeans 5.5, as well as its additional packs (e.g., Enterprise Pack with UML/BPEL/SOA and Visual Web Pack for JSF Development), is available as both ZIP downloads as well as cross-platform InstallShield installers. Under Windows, the installer integrates seamlessly into the OS, including registering the proper desktop shortcuts and adding an uninstaller in the Add/Remove Programs panel. Under Linux, it simply installs into the specified directory and creates a startup icon on the GNOME or KDE desktop. Unfortunately, it does not come packaged as an RPM or a .deb file, nor does it offer a standard repository, which would allow Linux users to install it as they do any other application.

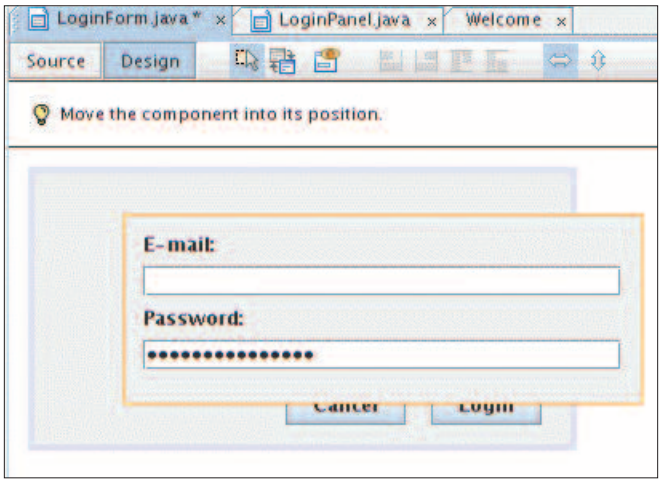
General Features

At one time, NetBeans was synonymous with everything that was wrong with Swing: slow, bloated, ugly, and just plain unpleasant to work with. However, the NetBeans team has performed a massive overhaul of the entire IDE starting with version 5, and the combination of NetBeans 5.5 and JDK 1.6 provides arguably a top-notch user experience, in particular under Windows (Linux still has some UI glitches that are supposed to be addressed in NetBeans 6.0. In particular, version 5.5 lacks native GTK look and feel support).

The windowing system is about as advanced and flexible as one could imagine, with the ability to dock/hide/swap nearly any panel/editor in any possible configuration with great ease. I also found the menu layout very logical and easy to use, with all the most common functions being easily accessible (e.g., maintaining user libraries). All the while, overall stability and performance were excellent.

The basic Java editor is decent though definitely not the best in the field (in particular, code completion is somewhat slower than its competitors), but it is very workable. A basic set of refactoring functionality is also provided (in particular, the most commonly used rename/move features), although in my testing I found it often somewhat dangerous to use because it is not always context-aware. For example, renaming the package of an Action class in a Struts project does not update the corresponding entry in struts-config.xml (although it worked fine for refactoring JSF backing beans and updating their entries in faces-config.xml, as well as J2EE 1.4 sessions beans and their entries in ejb-jar.xml).

Figure 1. Matisse with Custom JPanel Components



Swing Development

NetBeans's crown jewel is its famous new Matisse GUI designer, based upon the new GroupLayout layout manager, which originally was developed by the NetBeans team itself as an extension prior to being included in the base JDK. The combination of baseline support (i.e., the ability of controls to align automatically based on the position of the actual text within a control) makes creating professional looking UIs very easy. In fact, the powerful resizing and anchoring functionality make this the best UI designer I have seen for any language on any platform.

After installing version 5.5, I also grabbed the latest set of updates that were back ported to 5.5 from the newer 6.0 builds. These included some very productive features, such as automatic internationalization (with control per each form/dialog/panel as to which ResourceBundle contains all the internationalized strings), as well as the ability to use custom forms/panels (as long as they have been compiled at least once within your project). Matisse is a fine example of what the NetBeans team can do when they are at their best. (See Figure 1 for a sample of Matisse in action.)

Figure 2. NetBeans Web Application Wizard

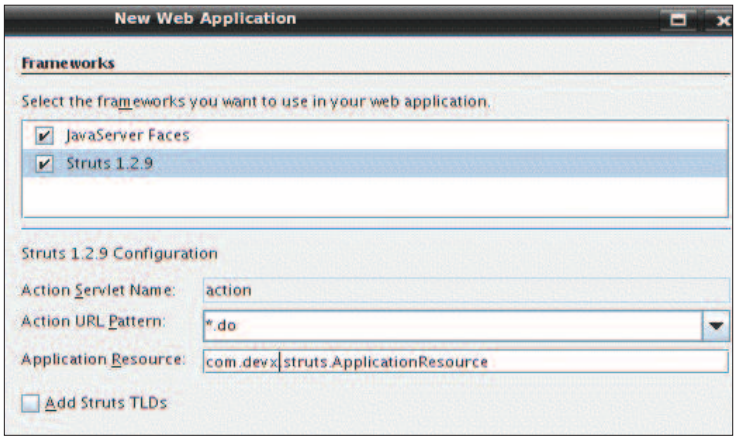
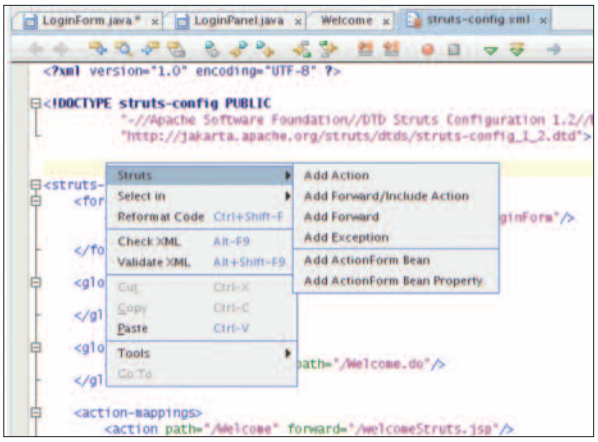


Figure 3. Struts Support in NetBeans



For Swing developers, an additional benefit of using NetBeans is the ability to use its very powerful RCP (Rich Client Platform) wizards for creating new, sophisticated Swing applications. In light of the popularity of Eclipse's SWT-based RCP, the NetBeans team has gone to great lengths to provide a viable Swing-based alternative and, dare I say, with impressive results (although the initial learning curve might be a bit steep for smaller projects).

JSP/Struts Development

NetBeans comes with a good JSP editor with all the basic features that one would expect and all the basic wizards and plumbing to start a new Struts project. The inclusion of an embedded Tomcat container makes developing and testing JSP apps particularly easy and fast. (Figure 2 shows the NetBeans Web application wizard.)

The NetBeans Web application wizard automatically configures Web.xml and struts-config.xml and enables adding Tiles and Validator support. However, beyond that the only support it provides are some context menu options in struts-config.xml and wizards to add ActionForms, Actions, and Forwards. It offered no visual editors to show the page navigation within the application and provided absolutely no additional support for configuring Tiles and Validators (outside of creating the original configuration files and including the required libraries).

Personally, I found Struts support in 5.5 quite decent but definitely not as polished as what Matisse delivers. Also, no visual editor for JSP or HTML pages is available (not an issue for power coders, but a nice feature for more junior programmers). The lack of proper embedded JavaScript support for JSP editors along with only a basic CSS editor (though much better if the Visual Web Pack is installed) are areas that could use some improvements as well. (See Figure 3 for a sample of NetBeans's Struts support.)

JavaServer Faces Development

The NetBeans JSF support is pretty much identical to its Struts support. It provides a wizard to get a basic project started, includes the libraries, configures all the required files, and even provides code completion for backing beans' properties in the JSP editor, as well as a few wizards for navigation rules in faces-config.xml. No support for Apache MyFaces is available (only JSF RI), so the initial project setup has to be hacked by hand to swap out the JSF RI and use MyFaces.

No visual editors are provided whatsoever to maintain the faces-config.xml file. Everything is pretty much done through raw XML editing or the two or three basic context menu wizards. While this is workable and definitely sufficient for power coders, it hardly achieves the ease of use or productivity that other IDEs deliver for JSF developers.

However, the NetBeans Visual WebPack offers an alternate solution for building JSF applications by porting most of the features from Sun Java Studio Creator directly into NetBeans. This includes a Matisse-style GUI builder for Web pages, with a rich set of JSF controls (recently open-sourced as Project Woodstock) that extend the basic JSF RI set, as well as support for data binding (both directly against database queries and even via JPA/Hibernate).

Despite all of these first-class features, the Visual Web Pack has a few drawbacks that might stop many Java shops from using it, namely:

1. It lacks support for page templates (although you can save a Page as a template and use it as a base for new bases, but obviously changes to it do not cascade down). Support for either Facelets or the Tiles support from MyFaces would be of great use here.

2. It lacks support for Apache MyFaces (potentially an issue when deploying to app servers that use MyFaces RI instead of Sun's implementation, such as JBoss).

3. It has inflexible auto-generation of backing beans (which basically mimic the structural context of each JSF page in a straight ASP.NET fashion, making it unusable with conversation-oriented backing beans such as the ones required by JBoss Seam).

A great-to-have feature would be support for Facelets, which many in the JSF community are pushing as the total replacement for JSP (in particular, the JBoss Seam team has openly campaigned for it in all of its documentation).

Though if the NetBeans team addresses these deficiencies in the next release, I can't imagine any tool being as productive as the Visual Web Pack for JSF development.

Enterprise Development

NetBeans 5.5 was the first IDE to offer support for JPA and EJB 3.0, and it delivered quite well in this area. In particular, the auto-generation of JPA entity classes with annotations (including proper setup of the persistence.xml file, even with basic connection information) has saved me countless hours of boring, repetitive hand coding. The auto-generated JPA code is very high quality and instantly usable. Frankly speaking, developing a Swing app with Matisse and querying/updating a database via JPA was the first time my productivity has reached the level I worked at in PowerBuilder many years ago (which I still remember as the most productive client/server UI development tool ever, even though it had more than its share of imperfections).

For pure enterprise development, NetBeans offers both top-notch J2EE 1.4 and Java EE 5 support. In particular, for J2EE 1.4 projects, the EJB wizards take care of generating all the required code (including the business/home/remote interfaces, as well as their stub implementations – not to mention wiring the ejb-jar.xml configuration file). In short, NetBeans takes care of most of the verbosity related to J2EE 1.4 and provides polished out-of-the-box support for Java EE 5 as well (the first IDE to do so, although IDEA followed quite quickly).

On the down side, NetBeans offers official support only for deploying enterprise applications to GlassFish/Sun Application Server and JBoss. Nonetheless, the update site offers additional plugins for WebLogic and WebSphere, although I am not sure if the NetBeans team officially supports them.

The free Enterprise packs also add powerful UML diagram features (including two-way editing and synchronization with Java code), as well as BPEL/SOA editors. However, evaluating these was outside the scope of this review.

Suggestion to NetBeans

I really like NetBeans. Its underdog team came out fighting from a position where everyone thought Eclipse would eat it alive, and it has delivered some amazing features in its latest release (and much more to come in NetBeans 6.0). However, it simply does not have the same size community that Eclipse does. This starts to detract from the product when it comes to features that it does not offer out of the box (e.g., Google Web Toolkit, JBoss Seam, Jasper Reports, Spring, Tapestry, XDoclet, Echo2, Apache MyFaces, etc).

One of the smart things IBM did early on with Eclipse was give up control and create an Eclipse foundation, which has attracted a lot of third-party developers. Maybe it's time Sun gave up its iron grip on NetBeans as well and created an equivalent NetBeans Foundation. Eclipse has no technical advantage whatsoever over NetBeans at this point (frankly speaking, I think NetBeans looks more like a native Windows application than Eclipse does, despite the whole Swing vs. SWT debate). NetBeans simply needs more resources behind it to become number one in the Java IDE space, and I don't think Sun can make that happen on its own.

IntelliJ IDEA 6.0.4

Vendor: JetBrains
Website: www.jetbrains.com/idea
Price: $499.00 ($299.00 upgrade)

Installation
Under Windows, IDEA provides a regular EXE installer. Under Linux, the installation is much more Spartan, consisting of a single TAR.GZ file. Upon extracting the contents, you have to manually change to the "bin" subdirectory and execute "./idea.sh". It fails if you do not have the $JDK_HOME variable set up correctly, preferably in your .bashrc file as follows, for example:

```
JDK_HOME=/home/jacek/Dev/Java/JDK/jdk1.6.0_01
export JDK_HOME
JAVA_HOME=/home/jacek/Dev/Java/JDK/jdk1.6.0_01
export JAVA_HOME

export PATH=$JAVA_HOME/bin:$PATH
```

Unfortunately, JetBrains does not provide a standard .deb or RPM file for any of the major Linux distributions. Therefore, the installation does not integrate into the desktop very well (e.g., no desktop shortcuts or K Menu entries are created). Worse, double clicking on the "bin/idea.sh" file from the Konqueror file manager did not work either (it was returning an error about JAVA_HOME not being set up, even though it was actually set up correctly). The only option that seemed to work was to drop to command line and execute "./idea.sh" manually. The IDEA installation experience under Linux needs some serious polish. It was the weakest of the three tested IDEs.

General Features
Just like NetBeans, IDEA comes with a very flexible layout, providing the ability to dock/pin/float panels in a variety of configurations. However, it didn't seem quite as smooth and configurable as the NetBeans windowing system (which IMHO is the one to beat).

However, this is a minor gripe in light of the outstanding IDEA editor, which has long been regarded as its crown jewel. The editor is fast, with complex context-sensitive color highlighting, hints, and suggestion pop-ups – not to mention an impressive array of refactoring options. The IDEA editor is any hardcore coder's dream. The more time I spent in it, the more little touches I found that made programming that much more efficient. I'm sure I barely explored all of its functionality during the limited time I had for reviewing. (See Figure 4 for a sample of IDEA's editor and its advanced coloring/syntax highlighting.)

Swing Development
IDEA provides a fairly powerful GUI editor. It doesn't quite live up to NetBeans Matisse's, but it arguably is the next best thing. In particular, its support for JGoodies Forms (arguably the best layout manager available prior to GroupLayout) places it well ahead of those that are still stuck on GridBagLayout (like Eclipse's Visual Editor).

Interestingly, IDEA keeps the generated UI layout in a separate ".form" file (similar to NetBeans), but it does not generate the corresponding Swing code by default. It gets generated only during compilation via a proprietary GUI compiler (which can also be packaged as a separate Ant task). If you prefer to have the IDE-generated code directly

in your .java file, you have to enable it via an option (which in my opinion is the preferable approach, since I would prefer not to have all of my UI code hidden and available only after running a IDE-specific code-generation mechanism).

Figure 4. IDEA Editor in Action

```
/**
 * Method generated by IntelliJ IDEA GUI Designer
 * >>> IMPORTANT!! <<<
 * DO NOT edit this method OR call it in your code!
 *
 * @noinspection ALL
 */
private void $$$setupUI$$$() {
    panel1 = new JPanel();
    panel1.setLayout(new FormLayout("fill:max(d;4px):noGrow,left:4dlu:noGrow,fill:513px:noGrow", "center:d:noGrow,top:4dlu
    panel1.setBorder(BorderFactory.createTitledBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8), null));
    final JLabel label1 = new JLabel();
    label1.setText("Email address:");
    CellConstraints cc = new CellConstraints();
    panel1.add(label1, cc.xy(3, 1));
    textField1 = new JTextField();
    panel1.add(textField1, cc.xy(3, 3, CellConstraints.FILL, CellConstraints.DEFAULT));
    final JLabel label2 = new JLabel();
    label2.setText("Password:");
    panel1.add(label2, cc.xy(3, 5));
    passwordField = new JPasswordField();
    panel1.add(passwordField, cc.xy(3, 7, CellConstraints.FILL, CellConstraints.DEFAULT));
    final JPanel panel2 = new JPanel();
    panel2.setLayout(new FormLayout("fill:d:noGrow,left:4dlu:noGrow,fill:max(d;4px):noGrow", "center:max(d;4px):noGrow"));
    panel1.add(panel2, cc.xy(3, 9, CellConstraints.RIGHT, CellConstraints.DEFAULT));
    button2 = new JButton();
    button2.setText("Button");
    panel2.add(button2, cc.xy(3, 1, CellConstraints.RIGHT, CellConstraints.DEFAULT));
    button1 = new JButton();
    button1.setText("Button");
    panel1.add(button1, cc.xy(1, 9));
    JPasswordField passwordField1;
}
```

JSP/Struts Development

Struts support in IDEA is nothing short of outstanding. In a fine example of the sort of attention to detail that IDEA is known for, it can even download all the required Struts libraries for you! Not only does it automatically set up all the configuration files (including Tiles and Validator), but it also provides a dedicated Struts panel called simply Struts Assistant, which provides graphical editors and productivity wizards for all Struts configuration files, including tiles-config.xml and validation.xml.

IDEA has no visual editor for JSP/HTML pages, but the regular IDEA JSP/HTML editor is superb even without a visual component. The support for embedded JavaScript (with full code completion!) especially will be a godsend to anyone dealing with large amounts of DHTML or AJAX code. As another example of the type of attention to detail others can only dream of, the JavaScript editor comes with support for browser-specific elements (IE, Mozilla, and Opera), as well as popular AJAX frameworks such as Dojo, Bindows, and Prototype. (See Figure 5 and Figure 6 for samples of IDEA's Struts support.)

For hardcore, cutting-edge Web 2.0 development, IDEA delivers full-blown support for GWT (Google Web Toolkit) as one of its core features. I find that very impressive, especially considering how new GWT is (but since it is sponsored by Google, the odds are it will be a winner and it's good to see IDEA supporting it so early).

Figure 5. IDEA Struts Web Application Setup

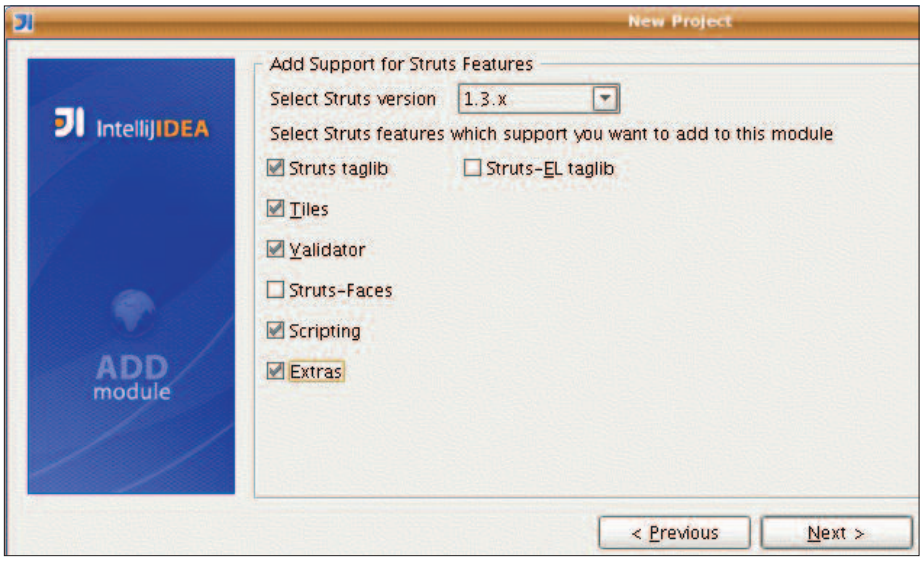


Figure 6. IDEA Struts Assistant

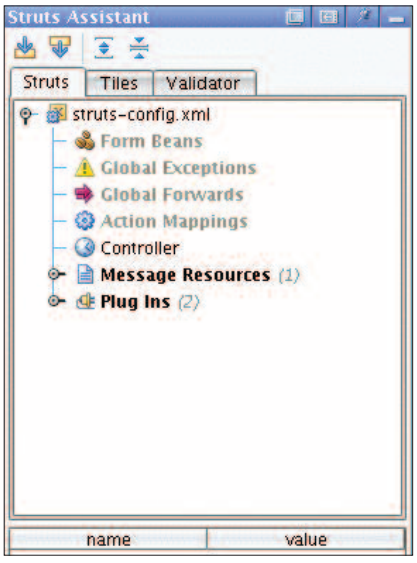


Figure 7. IDEA JSF New Web Application Setup

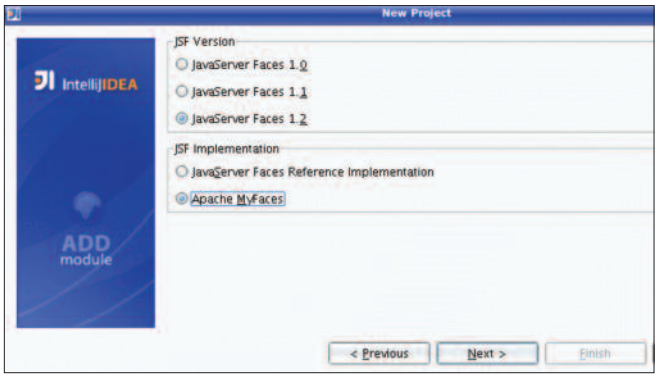
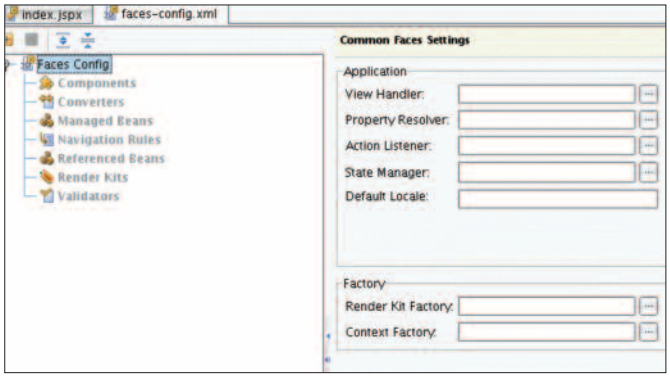


Figure 8. IDEA JSF Configuration



JSF Development

Just like its Struts counterpart, the JSF project options are an exercise in flexibility: you can choose which JSF version (1.0, 1.1, or 1.2) and implementation (JSF RI or Apache MyFaces) you're going to use, and then IDEA can even download the required libraries for you (and it displays productivity hints while downloading the JARs – now that's attention to detail). NetBeans team take note: this is exactly the kind of broad support for all popular open-source frameworks or implementations (instead of just Sun-sponsored ones) that is missing in your IDE. Also, the JSF module provides out-of-the-box support for JBoss Seam, which by all accounts seems to be on the way to becoming the standard framework for JSF development (similar to the way Struts was for JSP). This is further proof that IntelliJ is well aware of cutting-edge development in the Java Web world.

Refactoring seemed fully JSF-aware (e.g., moving a managed bean to a different package properly updated faces-config.xml). (See Figure 7 and Figure 8 for samples of IDEA JSF support.)

Enterprise Development

IDEA provides thorough support for the J2EE specifications. More importantly, it provides full-blown support for EJB 3.0 and JPA, although not quite as well as NetBeans yet (which automatically adds entries for the JPA provider in the persistence unit and generates code for named queries on all entity fields, something that I found missing in IDEA after getting used to it in NetBeans 5.5). IDEA does come with the option to view the ER Diagram for a JPA Persistence Unit, but unfortunately this seems available only in an EJB module. When using JPA in a regular Web module, I was not able to invoke the ER Diagram option. Aside from this minor gripe, IDEA's overall J2EE/Java EE 5 support is top notch. It even offers an upgrade path from J2EE to the annotations-based approach of Java EE 5!

As far as application servers go, IDEA provides deployment plugins for all the major players, namely WebLogic, WebSphere, JBoss, Geronimo, and Glassfish.

For unit testing, it supports JUnit4 and provides an integrated tool for measuring code coverage as well.

Eclipse 3.2.2 "Callisto"/ MyEclipse 5.1.0 GA

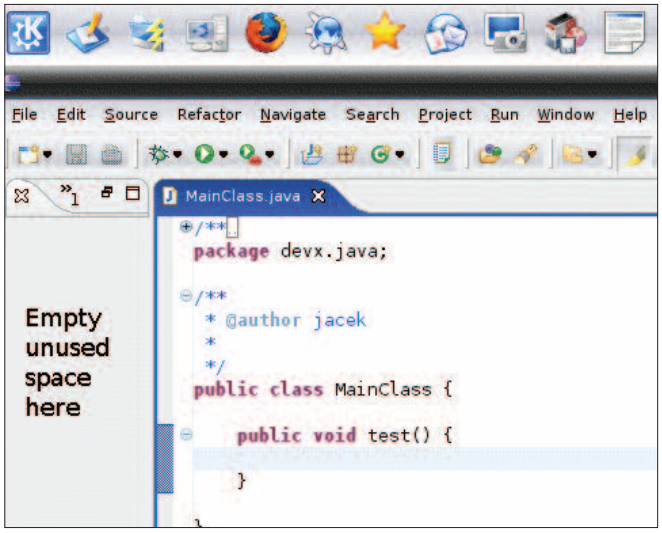
Vendor:	Eclipse Foundation	**Vendor:**	Genuitec
Website:	www.eclipse.org	**Website:**	www.myeclipseide.com
Price:	Free (base distribution)/Open-Source	**Price:**	MyEclipse IDE
		Distribution:	$49/year subscription

Installation

On Windows and Linux the base Eclipse distribution is just a simple .zip or .tar.gz file that you extract in whatever directory you deem necessary. This provides you with a bare-bones IDE that is basically capable of creating a "Hello World" program, but not much else. In order to turn Eclipse into a workable environment, one has to

download extra plugins from the Eclipse Website (available directly from within Eclipse via Help -> Software Updates -> Find And Install). The ones I was most interested in were the Visual Editor (for SWING GUI building), the Web Tools Project (for JSP support), the JSF Tools, and Dali (for JPA support). The last two are officially declared as "preview" and not yet at version 1.0.

Figure 9. One of Eclipse's Unusually Collapsed View



When it comes to installing new plugins, Windows was straightforward. Under Linux, it is possible to download Eclipse from a standard repository (most Debian-based or RPM-oriented distros feature Eclipse), which integrates perfectly with the way Linux applications are usually installed. However, this method installs Eclipse into a system directory (e.g., "/usr/lib/eclipse" under Ubuntu/Kubuntu), which can be updated only if running as root or with root privileges via 'sudo'. Unfortunately, Eclipse seems unaware of this and downloading the plugins ended in an error since Eclipse did not prompt me for the root password when attempting to install them under the restricted "/usr/lib/eclipse" folder. It would be great if Eclipse enhanced this little detail in a future release. As a simple solution, I just copied the entire local Eclipse installation to a folder in my home directory and was able to install all the additional plugins without any further issues (I guess I could have just logged into a session as "root," but I prefer to avoid doing that).

MyEclipse comes with a Java-based installer that under Windows integrates perfectly with the OS (including shortcuts) and does an acceptable job under Linux (although it does not create any desktop shortcuts).

General Features

When Eclipse appeared on the scene, its amazing, fast, and feature-rich Java editor was quickly recognized as its crown jewel. Among its attributes were:

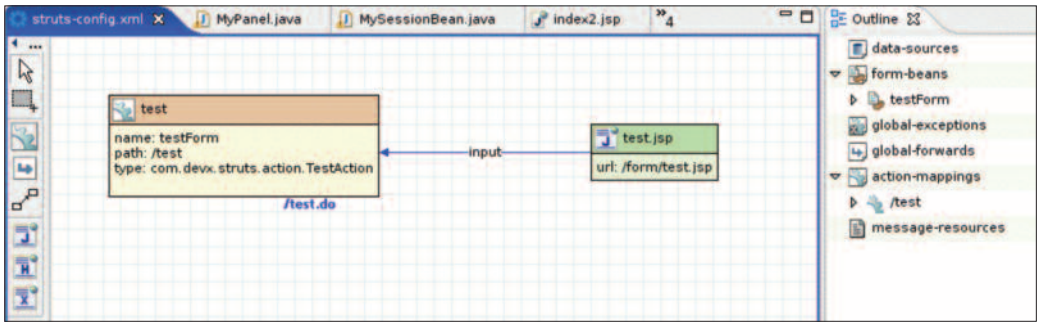
- Fast performance
- Powerful refactoring
- Quick fixes for errors
- The ability to fix/organize imports
- Lots of polish seen in little details (e.g., attractive Javadoc pop-ups on code completion).

It indeed is a top-notch editor in every meaning of the term.

I was not as fond of Eclipse's windowing system, in particular its perspectives/views paradigm (I hated this same system back in NetBeans 4.0 as well, but they were wise enough to replace it). I very much prefer the simple approach of having all the relevant editors/palettes/panels configurable in a single window, without the confusion of perspectives. Admittedly, that is a matter of personal preference though.

I also found some aspects of the windowing system to be illogical. For example, minimizing the "Package Explorer/Hierarchy" view does not collapse it to the side (as I would expect from working with other IDEs, be it NetBeans or even Visual Studio.NET). Instead, Eclipse just folds it up and leaves a large portion of the screen unused – a very unusual design decision. (See Figure 9 for an example of this strange windowing behavior.) Besides that, the overall windowing system is quite capable, but I prefer NetBeans's system much more.

Figure 10. MyEclipse's struts-config.xml Editor



Swing Development

The Visual Editor in Eclipse was by far – at least a mile – the weakest GUI builder of any of the major Java IDEs, mostly because the most advanced layout it supports is GridBagLayout. After working with NetBeans's Matisse, I could never imagine myself going back to that ancient and cumbersome method of creating Swing UIs. Fortunately, in what is a testament to the power of the community that has grown around Eclipse, MyEclipse delivers the Matisse GUI builder integrated directly into Eclipse! And it works very well, although obviously it will always be a few updates behind the cutting-edge enhancements that the NetBeans team is continuously adding to it (such as the bean binding feature that is supposed to be delivered as part of NetBeans 6.0). Nevertheless, the mere fact that Matisse is available on the Eclipse platform (even if it is part of a commercial solution) highlights why it is the 800lb. gorilla in the Java IDE space: none of its competitors have a similar community of plug-in developers that are willing to complement (or sometimes replace entirely) Eclipse's base functionality.

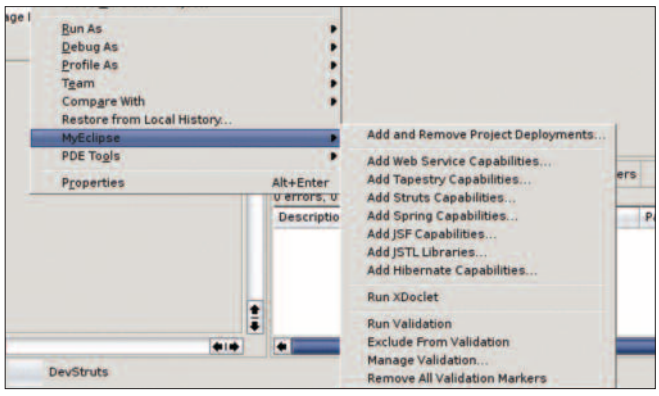
JSP/Struts Development

The Web Tools Project (WTP) adds Web-development features to Eclipse. It is quite capable for basic JSP development, offering a solid JSP editor. However, deploying your Web application seems to be somewhat flaky. It often locked up with errors when I was deploying it to Tomcat. This was in direct contrast to the other IDEs, which did not exhibit any of these issues.

The WTP also lacks any built-in support for Struts development, which seems to be a major hole in its functionality (although a quick search at www.eclipseplugincentral.com revealed no less than 24(!) plugins for Struts support, both commercial and open-source). To no surprise, MyEclipse fills this hole quite thoroughly with full Struts support, including graphical editors for struts-config.xml, as well as wizards for creating new Actions and FormBeans (Figure 10 shows its Struts editor in action.).

Eclipse with MyEclipse added was also the only tool to provide a visual editor for creating JSP/Struts pages. (See Figure 11 for a sample of the capabilities MyEclipse can add to a Web project.) Unfortunately, I was not able to test it thoroughly due to an Eclipse bug that disabled it under Linux.

Figure 11. MyEclipse's Web Project Options



JSF Development

The base Eclipse distribution provides a preview version of its upcoming support for JSF development. Despite its "preview" status, it is actually already quite usable and, dare I say, better than some of its competitors' supposedly more mature JSF functionality. This includes a visual editor for the faces-config.xml, code completion in JSP pages for managed beans' properties, new managed bean wizards, as well as visual editors to set up

Converters, RenderKits, and Validators. I am excited to hear that the Eclipse team is planning to release a visual JSP/JSF editor as well. MyEclipse's JSF functionality is very similar in scope, but in my testing the base Eclipse JSF support was more than adequate.

Enterprise Development

Eclipse does not provide official support for JPA yet. It does have an incubation project called Dali that is geared to deliver this functionality, but it's still in the fairly early stages of development. Although I was able to generate a working set of entities from a database with it, however, even though the generated code wasn't quite at the level that NetBeans currently generates (e.g., it did not have any named queries created automatically). If your shop has not moved to JPA yet and is using straight Hibernate instead, then consider MyEclipse an option as it has quite a rich level of Hibernate support.

Figure 12. MyEclipse's EJB Code Generation

For J2EE development, Eclipse supports creating EJB and EAR modules, although in order to avoid J2EE's complexity it seems to be focused on generating EJBs via XDoclet, which admittedly was the best solution available before Java EE 5 and EJB 3.0 dramatically reduced enterprise application complexity. MyEclipse extends this functionality by providing additional wizards for session beans, message-driven beans, and container manager persistence beans, also driven by XDoclet.

Neither Eclipse nor MyEclipse currently seem to have Java EE 5 support, but considering it is still a fairly new specification I presume the Eclipse Foundation is busy adding this for a future release. (See Figure 12 to see MyEclipse's J2EE 1.4 EJB wizards.)

IDE Strengths by Development Area

I will be the first one to admit that no review is perfect. The respective team behind each IDE probably could make counterarguments to the many points in this article. It is simply impossible to evaluate all the possible development needs (and this review barely covered topics such as RCP, UML, JUnit, and reporting support), so obviously your choice of IDE should be based heavily on the particular Java technology with which you are most comfortable.

Each of the IDEs reviewed here can do an admirable job in pretty much every facet of Java development. However, some are better than others, depending on whether you are doing Swing, Web, or enterprise development. So I organized the review summary into these subject areas.

Swing Development

If your shop specializes in Swing development, NetBeans is definitely the way to go. Matisse is simply way ahead of the competition. If for corporate reasons you have no choice but Eclipse, then I definitely suggest MyEclipse with its Matisse4Eclipse builder. After those two choices, I would rate IDEA (due to its support for JGoodies Forms) next and Eclipse's default Visual Editor dead last – way behind any competition. It should simply be avoided, period.

JSP/Struts Development
Things are a lot more heated here. I would give a clear advantage to IDEA, followed by MyEclipse, and then NetBeans. Due to lack of build-in Struts support, the base distribution of Eclipse isn't much of a contender.

JSF Development
The three are in quite a tight race in this category as well. Once again, I feel IDEA comes out on top here, followed closely by Eclipse/MyEclipse, and the basic support offered by NetBeans in last place. Admittedly, this ranking would look a lot different if you take the NetBeans Visual Web Pack into consideration (assuming its limitations are acceptable), which would move it into the front of the pack.

Enterprise Development
For JPA support, I would rank NetBeans first (simply due to the quality of its generated code and support for properly setting up the persistence units), followed by IDEA, and lastly the still limited functionality of Eclipse's Dali project. If you are willing to abandon the standard JPA approach and accept straight Hibernate as an alternative, then MyEclipse becomes a worthwhile contender as well.

For enterprise development, I'd say IDEA wins out with its rich support for both J2EE and Java EE 5, followed closely by NetBeans (which also does an impressive job here), and last is Eclipse/MyEclipse (mostly due to their current lack of support for Java EE 5).

Ignore .NET at Your Own Peril

If Eclipse is the 800lb. gorilla of Java IDEs, Microsoft is a menacing 10-ton King Kong somewhere in the background. As someone who has done a lot of work in C#/.NET in the past few years, I keep quite up to date on what Microsoft is doing in .NET 3.0 and its next version of Visual Studio (codenamed "Orcas"). I hope none of the Java IDE vendors are getting too comfortable and resting on their laurels, because Microsoft is putting a massive amount of R&D effort in both libraries and development tools. Thus, the Java ecosystem can remain healthy only if it can match that (or even better, exceed it as in my humble opinion it has done with Matisse, JPA, and EJB 3.0).

Companies and technologies that have ignored Microsoft's impact have usually ended up in the dustbin of IT history (and I write that as an ex-Sybase/PowerBuilder developer who has seen a once great tool mercilessly crushed under the weight of both its own mistakes and Microsoft's seemingly never-ending resources). I am very glad to see that the mistaken old Java mentality of "release the APIs first and then wait for the development tools of varying quality to appear much later" is being aggressively replaced with "release the APIs and world-class development tools for them as soon as possible", since that is exactly what Microsoft has been doing for years. ■

Jacek Furmankiewicz *is a Senior Developer/Designer at Compuware Corporation of Canada. He has 12 years of professional IT experience, ranging from UNIX, PowerBuilder, C#/Microsoft .NET, Java, PHP, as well as Microsoft SQL Server and Oracle.*

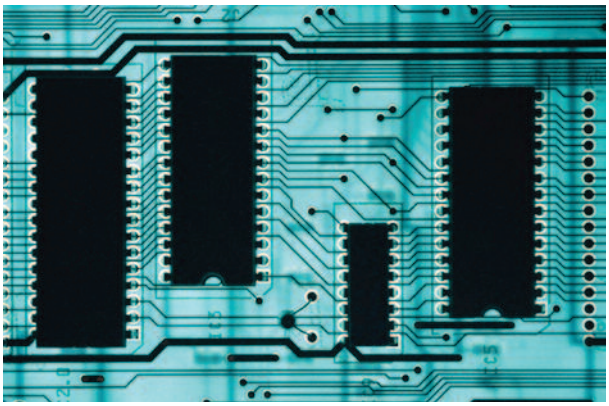
This content was adapted from DevX.com's Web site.

The Work Manager API: Parallel Processing Within a J2EE Container

The Work Manager API offers a solution for performing parallel processing directly within a managed environment. You can leverage this offering to implement parallel processing within a J2EE container

by Rahul Tyagi

Corporations encounter various scenarios in which they need parallel processing to achieve high throughput and better response times. For example, various financial institutions perform reconciliation as batch jobs at the end of each day. In these cases, a company may need to process millions of units of work to reconcile its portfolio. These units of work typically are processed in parallel. This article demonstrates how to accomplish parallel processing within a J2EE container.



Jupiterimages

The J2EE design revolves around the request/response paradigm. For a login request, a user typically provides a user name and password to the server and waits for a response to get access to the site. A J2EE container can serve multiple users at the same time (in parallel) by managing a pool of threads, but for various reasons opening independent threads within a single J2EE container is not recommended. Some containers' security managers will not allow user programs to open threads. Moreover, if some containers allowed opening threads, then they would not manage those threads and therefore no container-managed services would be available for the threads. That would leave you to implement these services manually, which is tedious and liable to add complexity to your code.

Until relatively recently, performing parallel processing directly within a managed environment (J2EE container) was not advisable. Thankfully, IBM and BEA came up with a joint specification that resolves this problem. This JSR is named "JSR-237: Work Manager for Application Servers". JSR-237 specifies the Work Manager API, which provides abstraction from the lower-level APIs that enable an application to access a container-managed thread. Work Manager API also provides a mechanism to join various concurrent work items, so an application can programmatically add the dependency of work completion as a condition for starting other tasks. This can be useful for implementing workflow types of application. These features were difficult to implement prior to Work Manager.

This article discusses this specification and its design, and presents some code snippets for implementing a concurrent application for a managed environment. First, it discusses the design of the Work Manager API and its key interfaces.

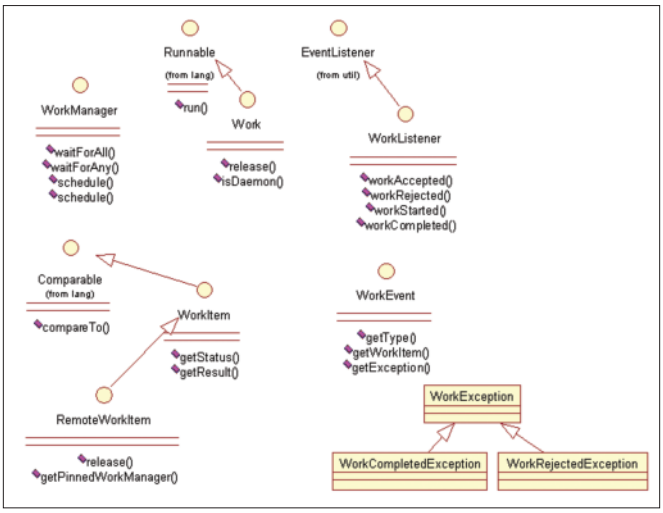
Work Manager Design

The Work Manager API defines Work as a unit of work, which you want to execute asynchronously. Work is an abstraction of the lower-level java.lang.Runnable interface. Implementation of the Work interface requires you to define a run() method and implement business logic for performing tasks/work in this method. The Work Manager API has six key interfaces for implementation:

- WorkManager
- Work
- WorkListener
- WorkItem
- RemoteWorkItem
- WorkEvent

Figure 1 shows a class diagram for these interfaces.

Figure 1: Work Manager Class Diagram

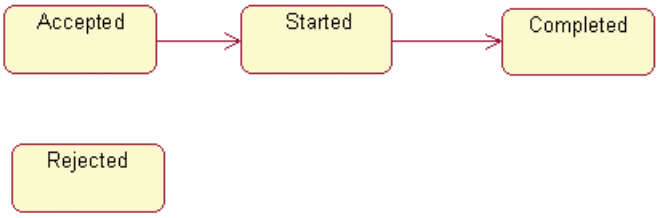


J2EE container providers such as BEA and IBM must implement the WorkManager interface, and server administrators can create WorkManager by defining it in their J2EE container configurations. Most leading J2EE containers provide user interface (UI) tools for defining WorkManager. WorkManager encapsulates a pool of threads. By invoking the schedule(Work work) method of the WorkManager interface, a client can schedule work for asynchronous execution. Behind the scenes, the Work Manager implementation gets a container-managed thread for executing the Work object. So work is executed in parallel with the current thread. A previously mentioned, the Work interface implements a run method of the java.lang.Runnable interface, so a thread can execute an object of type Work.

Work Lifecycle

Work has states in its lifecycle, as illustrated in **Figure 2.**

Figure 2: Work Lifecycle



A Work lifecycle starts after the schedule method of Work Manager is invocated. At any given time, Work can be in any one of the following states:

- Accepted: Work Manager has accepted work for further processing.
- Started: Work just started execution on a thread. You can assume this state just prior to entering the run method of Work.
- Completed: Work just completed execution. You can assume this state to be just after exiting from run method.
- Rejected: Work could be rejected at various stages. Work Manager can reject work in case it's unable to process a request.

Work Manager provides a listener interface (WorkListener) that defines four callback methods associated with the Work lifecycle:

- workAccepted
- workStarted
- workCompleted
- workRejected

Work Manager invokes these methods at corresponding states of the Work lifecycle. The user can implement the WorkListener interface and pass implementation of WorkListener to Work Manager while scheduling work. Work Manager would invoke callback methods corresponding to each state.

Remote Processing of Work

A Work Manager implementation can execute work in a local JVM or transport work for execution by a remote JVM. To process work on a remote JVM implementation, the Work interface needs to implement the java.io.Serializable interface. A Work Manager implementation can check whether Work has implemented the Serializable interface, and then send it to a remote JVM for execution. Processing work in a remote JVM is implementation specific, and some implementations can execute work locally even if Work has implemented Serializable. Work Manager's design encapsulates the complexities of sending work to remote JVMs from a client.

App server providers could implement a scalable Work Manager framework by implementing a work manager capable of executing work on remote JVMs.

Adding Dependency to Work Completion

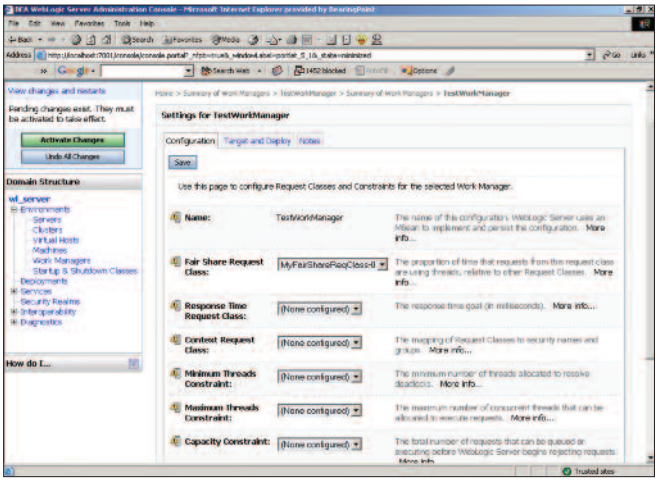
Work Manager provides a simple mechanism for adding dependencies of work completion on to other tasks. Work Manager provides a simple API for the common join operation. The current specification provides two methods for accomplishing a simple join. Both take a collection of WorkItem objects to check the status of Work objects and a timeout parameter to timeout after a specified interval, even if the condition is not met:

- waitForAll(): An application waits for all tasks to complete before moving on to further steps. This is a blocking call with configurable timeout value.
- waitForAny(): An application waits for any one of the work items to complete before moving on to the next step.

Work Manager defines two constants that correspond to timeouts:

- WorManager.INDEFINITE: Wait indefinitely for any/all work to complete (Use this option with care, because it could lead your application thread to wait forever.)
- WorManager.IMMEDIATE: Check status of any/all work completion and return immediately

Figure 3: WebLogic Administrative Console Showing TestWorkManager



Configuring Work Manager for J2EE Container

J2EE container providers that support the Work Manager API normally provide a UI-based tool for defining Work Manager. WebLogic 9.0, for example, provides such a tool for configuring Work Manager (**Figure 3** shows a WebLogic 9.0 administrative console showing a Work Manager configuration). The user needs to pro-

vide a work manager name and point Work Manager to a target server or cluster. As an example, I have created TestWorkManager.

An implementation also can provide features that are not part of a specification. For example, WebLogic allows users to configure Work Manager with the following optional parameters:

- Minimum thread constraint: The minimum number of threads allocated to resolve deadlock
- Maximum thread constraint: The maximum number of threads that can be allocated to execute requests
- Capacity constraint: Total number of requests that can queue or execute before WebLogic starts rejecting requests

Implementation of Work Manager-Based Application

Take the following steps to implement an application for parallel processing with WebLogic 9.0:

1. Implement the Work interface and define your business logic in the run() method. The current version of WebLogic 9.0 does not support execution of work in remote JVMs, but it may in the future. So it's a good practice to implement java.io.Serializable as well for future enhancements to Work Manager.
2. Implement the WorkListener interface for listening to work lifecycle events. Although not necessary, this step is a good practice.
3. JNDI lookup is the primary way of accessing Work Manager. Servlets and EJB can access Work Manager via JNDI lookup within the local JVM. You can associate the Work Manager resource to an application by defining resource-ref in the appropriate deployment descriptor. For a servlet to access Work Manager, you define the resource-ref in web.xml. For EJB, you can define the resource-ref in ejb-jar.xml:

```
<resource-ref>
<res-ref-name>wm/TestWorkManager</
res-ref-name>
<res-type>commonj.work.WorkManager</
res-type>
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</
res-sharing-scope>
</resource-ref>
```

4. Now, look at a code snippet that invokes Work Manager and schedules work for execution on a container-managed thread. The following snippet shows the code for using the Work Manager API:

```
//#1. Get Work Manager from local JNDI
WorkManager workManager;
InitialContext ctx = new InitialContext();
this.workManager = (WorkManager)ctx.lookup("java:comp/env/wm/TestWorkManager");

//#2. Create instance of Work and WorkListener implementation
Work work = new WorkImpl("HelloWorld");
WorkListener workListener=new WorkListenerImpl();

//#3. Schedule work for execution, which would start in parallel to current thread
WorkItem workItem = workManager.schedule(work, workListener);

//#4. Create list of WorkItem you want to wait for completion
```

```
26.  List workItemList=new ArrayList();
27.  workItemList.add(workItem);
28.
29.  //#5. Wait for completion of work
30.  this.workManager.waitForAll(workItemList, WorkManager.INDEFINITE);
31.
32.  //#6. You can get results from Work
     implementation
33.  WorkImpl workImpl= (WorkImpl)workItem.getResult();
```

In this implementation, you assume WorkImpl and WorkListenerImpl are implementations of the Work and WorkListener interfaces.

The Takeaway

This article discussed a simple and powerful API for parallel execution of tasks within a managed environment. It explained the high-level design of the Work Manager API and explored WebLogic 9.0's working implementation of the Work Manager API.

With the Work Manager API, developers can design robust applications for executing tasks in parallel, listen to work lifecycle events, and add the dependency of task completion to other tasks. Work Manager implementations also could be highly scalable if app server providers implemented a work manager capable of executing work on remote JVMs. ■

Rahul Tyagi *has more than 12 years of IT experience designing medium- to large-scale enterprise applications. He is also a member of expert groups for JSR-130 and JSR-241. Currently Rahul works as Technical Architect for BearingPoint Inc.*

This content was adapted from DevX.com's Web site.