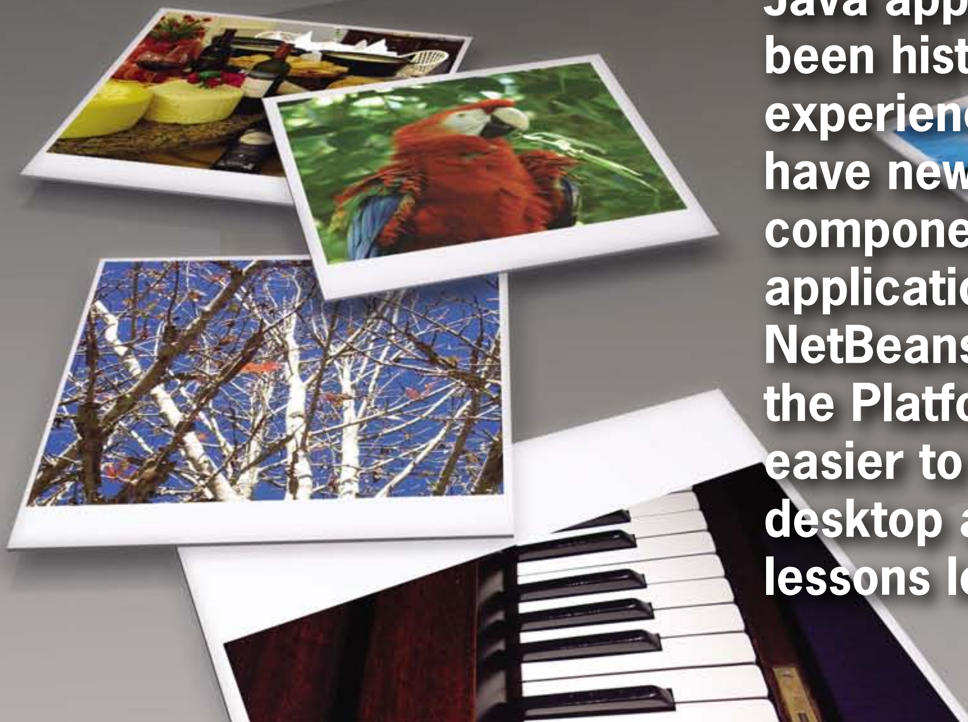
From Pain to Gain

Swing and the NetBeans Platform in the real world

Fabrizio Giudici



Developing rich desktop Java applications has been historically a painful experience; but now you have new advanced Swing components and a complete application framework in the NetBeans Platform. See how the Platform has made it much easier to develop a complex desktop application and the lessons learned in building it.

Like most people working with Java since its early beginnings, my first experience with the technology was with (small) desktop applications: some research stuff during my doctorate and a simple control panel for a healthcare call center. It was the age of AWT, and you really couldn't do much more. So I soon moved to the server side, where things appeared more robust and promising. They were indeed, and I stayed there for long and became a J2EE Architect.

A few years later, I was attracted to the desktop again because of a rising passion for digital photography. I still encountered many problems, but just before I threw the sponge, Sun and the developer community came to the rescue with SwingLabs, java.net and new versions of NetBeans. Now I am enjoying a (possibly) promising open-source application – blueMarine – which is based on the NetBeans Platform.

In this article, I'll tell you more about blueMarine's story and review some of the main NetBeans extension APIs. I'll show how these APIs are used and customized, while pointing out the problems I faced and how they were fixed. If you know just a little about NetBeans and you're involved with rich client applications, I think you will enjoy this article.

The beginning

The first time I ever wrote some Java code for managing my photos was around 2001, after getting bored with the OpenOffice spreadsheet I was using. I exported everything to XML and, by means of an XSLT transformation, defined my own database format which was managed by a very simple GUI based on Swing.

In summer 2003 I made the "big jump" into the world of digital cameras, buying a Nikon D100 (a professional SLR). It was one of the century's hottest summers in Italy, so I was forced to minimize the number of photo trips: walking outside was simply a pain. Being forced to stay at home, although in the relaxing environment of the Tuscany countryside, I spent most of my holidays studying the NEF format.

NEF is a "RAW file format" that at the time was mostly undocumented. A RAW file format holds the unprocessed data straight from the camera's CCD sensor and requires processing for being transformed into a quality picture; this processing is often seen as the digital counterpart of the old wet darkroom photo development. Having never owned a wet darkroom, I was intrigued about the possibility of "digitally developing" my photos, and started writing some Java code for this.

At the end of the summer I had created a simple thumbnail navigator with the capability of showing my own photos – blueMarine was born. A year later, the project had the capability of tagging photos with a catalog facility and of publishing galleries on the web.

However, I was irked by the fact that I needed more than a single piece of software to perform tasks such editing, printing, cataloging, archiving and web publishing. So I set about implementing all this workflow in a single application. Also, I decided it was high time I publicly released blueMarine, and so the first alpha release went on SourceForge under the GPL License (later changed to Apache 2.0). You can see one of the first versions in **Figure 1**.

Another force was pushing me on: the challenge of trying Java for digital image processing on a desktop computer. To me it was already evident that Java was good for scientific image manipulation; one example was that engineers at NASA were successfully using JAI, an advanced imaging API. But what about desktop processing for the casual photographer? To demonstrate that Java is good for a wide range of applications is something that I've always been pursuing since I started working as a Java consultant more than ten years ago.

The frustration

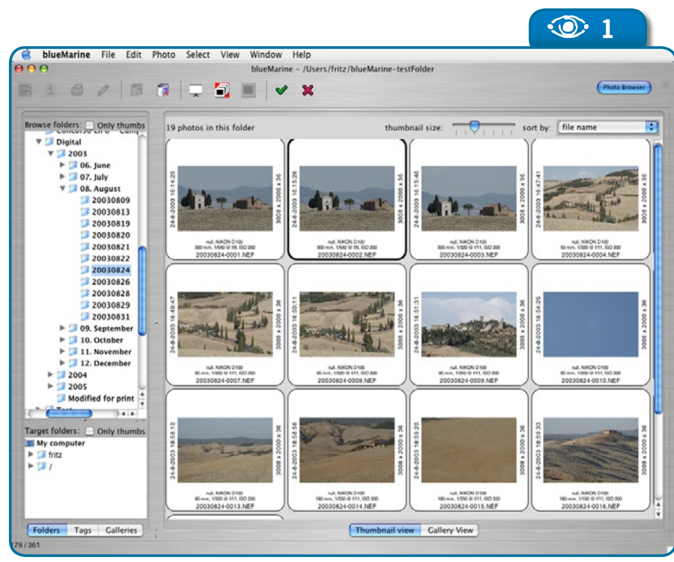
Notwithstanding the initial enthusiasm, at the end of 2005 I was pretty frustrated with the project. Performance wasn't much of an is-

The blueMarine Project

Aerith

Geertjan Wielenga's blog

Figure 1 The old blueMarine main window, developed on plain Swing.



sue but I was facing difficulties in developing a rich GUI application using plain Swing. Swing is an excellent API, but when you start using it to build a complex app you discover there's still a lot of stuff to add.

Implementing the missing pieces is no rocket science, but that work wastes a lot of time better spent elsewhere. Re-instantiate the problem for things such as building menus, having actions enabled in a context-sensitive fashion, defining a flexible and working docking mechanism for internal windows... and you'll find yourself spending most of your time writing generic GUI components, instead of working on the core of your application.

Until recently, there were few open-source libraries dealing with such issues, and most were unsatisfactory and cumbersome to integrate. There were also the early releases of NetBeans, but I was unsatisfied with their performance. Eclipse and SWT were an option, but I decided I wasn't really going to study a completely alternative and nonstandard API, with a very low learning ROI and a cumbersome way to integrate with Swing.

Summing up, I was seriously thinking about giving up with blueMarine – maybe Java wasn't yet ready for desktop development after all.

Romain Guy's blog
www.curious-creature.org

The Renaissance

However, there were a few concurrent events that saved the project: my participation at JavaPolis at the end of 2005, and the release of NetBeans 5.0 in early 2006.

At JavaPolis, I breathed the community atmosphere that I had mostly forgotten (three years had passed since my last JavaOne). This renewed my enthusiasm, which was piqued further by Romain Guy's presentation showing how effective GUIs can be built with Swing. I started looking at Romain's blog and by following links I got to other blogs such as Joshua Marinacci's, and from there to all the java.net and JavaDesktop stuff. I discovered there was a great deal of new interest in Swing; good quality Swing components such as at SwingLabs, cool demos – lots of material that I could use. But I still needed a platform.

A few weeks later, NetBeans 5.0 came out. The new release looked like it had finally fixed the traditional problems of the Platform, so I decided to give it a try. I started disassembling blueMarine, extracting only the imaging code and redesigning it to use the NetBeans Platform. After a few months, the first Early Access builds were ready to be delivered, and I had started using the tool for my own photo management. In the meantime, the zero-issues switch from my former PPC Apple iBook to the new Intel MacBook Pro was a strong sign that my choice had been right.

Today I'm working on making the new blueMarine stable and usable. New early access builds are available, and I'm running the required quality tests (the complete redesign obviously broke some of the stability of the previous release; that was the price to pay). **Figure 2** shows the Platform-based version of blueMarine at work.

The power of the NetBeans Platform

Now that you know the origins of blueMarine, I'll show an overview of the many benefits that NetBeans and Swing brought to

its development, as well as some problems I faced and how I fixed them.

First point: it's Swing!

To me, the fact that the NetBeans Platform is based on regular Swing is a huge advantage over competitors such as Eclipse RCP. If you look around, you can find a much broader choice of Swing components (including "cool" ones which implement animations and effects).

I concretely realized this advantage last June, when Joshua Marinacci released the source of an Aerith Swing component capable of rendering maps, named **JXMapViewer** (Aerith was one of the hottest demos at JavaOne 2006). I had been waiting for that moment for several weeks, as one of the features of blueMarine is geo-tagging (associating a geographical position to each photo so they can be shown over a map). Integrating **JXMapViewer** into blueMarine required just a few hours of work; you can see the result in **Figure 3**. The Swing choice was indeed rewarding.

The Module System

A NetBeans Platform application is naturally organized into modules – in fact, it's a set of modules bound together. Each module has a name, a set of version tags, its own classpath, and a list of declared dependencies. The developer can control which subset of the public classes is exposed to other modules, and the platform enforces the dependencies among modules (for instance, preventing a module to be installed if any of the required modules is not present or is too old).

Furthermore, an application can be extended at a later time by publishing new modules packed in *nbm* files, and users can set up their own "update center" for downloading updates from the Internet. Individual modules can be digitally signed and the system automatically pops up their license for approval if required.

The blueMarine project takes full advantage of this organization. The core APIs of the application are defined by a relatively small set of modules implementing a workspace manager, photo and thumbnail management and simple thumbnail and photo viewers. The more advanced features, such as the Catalog, the Gallery Manager, and geo-tagging functionality, including the Map Viewer, are implemented in separate and mostly independent modules that act as "clients" of the core APIs.

DataObjects, Nodes and ExplorerManagers

ExplorerManagers, **Node**s and **DataObject**s are probably the most

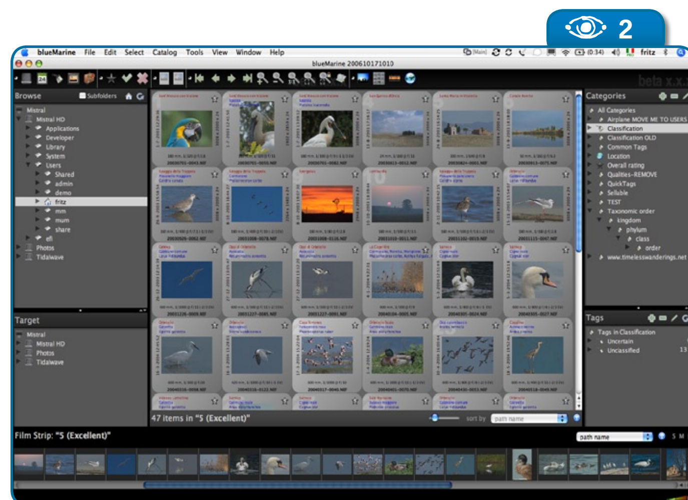


Figure 2
The main window of the new blueMarine application, developed on the NetBeans Platform.

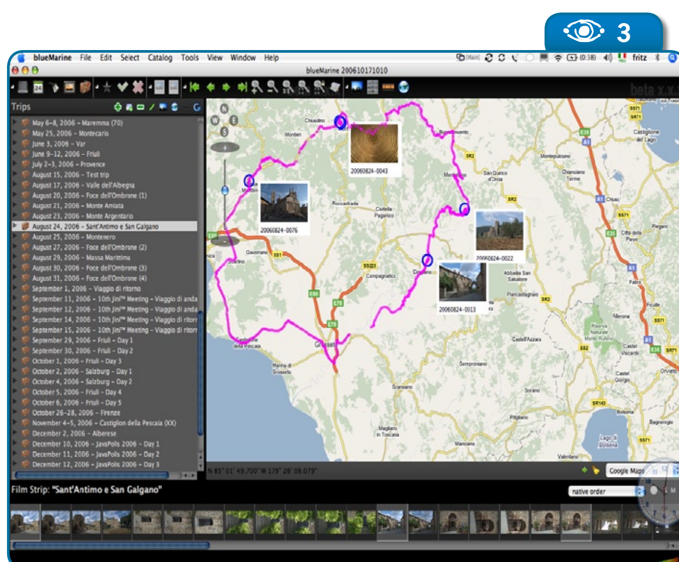


Figure 3
The Map Viewer with geo-tagged photos uses the *JXMapViewer* component.

useful APIs in NetBeans. With **DataObject**s you can implement your application-specific entities that are mapped to a file on the disk. For instance, blueMarine's basic entity is **PhotoDataObject**, which represents a photo in the database.

While **DataObject**s contain all the status and behavior of your entities, **Node**s can be bound to them for visualization purposes. They can also be aggregated in many different ways (as collections or graphs). The NetBeans Platform provides GUI components, such as tables and lists, which can use a set of **Node** objects as their model; among the most common are **BeanTreeView**, **ContextTreeView**, and **ListView**. Finally, an **ExplorerManager** controls selections and tree navigation.

Yes, this is nothing more than a sophisticated MVC implementation (see **Figure 4**), but an implementation where a lot of boilerplate code has been already written for you. For instance, the Platform APIs take care of things like drag-and-drop support (with such fine details as visual cues during the drag operation), cut-and-paste operations, and context menus.

The Lookup API

The NetBeans Platform components have a platform-controlled lifecycle (much like EJBs in a container), so they are not directly instantiated. In order to retrieve a reference to an existing module, you use the Lookup API. This API is very similar to other lookup mechanisms. You get a reference to an object starting from its "name", which is not a string but the corresponding **Class** object.

For instance, let's suppose we have a module called **it.tidalwave.catalog.CatalogImpl** (implementing an interface **it.tidalwave.catalog.Catalog**). First you "register" the module by putting a special file in the classpath, under the *META-INF/services* directory. The file must be named after the implemented interface and contain the name of the implementation class. Whenever a module is loaded, NetBeans scans for these special files, instantiates the objects and puts them into the "default" **Lookup** object, from where any other piece of code can later retrieve it.

I usually wrap the lookup code using the Locator pattern, as shown in **Listing 1**, and then perform lookups like this:

```
Catalog catalog = CatalogLocator.findCatalog();
```

This mechanism not only favors decoupling, but it also creates pluggable behavior. For instance, let's look at the map-rendering capability of blueMarine. As you know, there are many map providers around, such as Google Maps, Microsoft Visual Earth, NASA, and others. I want people to be able to extend blueMarine by plugging new code into it for handling additional map providers. The solution is simple: first define an interface – **MapProvider** – which declares all the required capabilities, then write alternate implementations, each one in its own module, e.g. **GoogleMapProvider**, **MicrosoftVisualEarthMapProvider**, etc.

Each implementation is registered in the default **Lookup** instance, using the same "name": **MapProvider** (multiple registered objects for the same name are allowed). Now, retrieving the objects becomes an easy task. An example is shown in **Listing 2**. You can add modules with new map providers, and the retrieval code will find them at runtime.

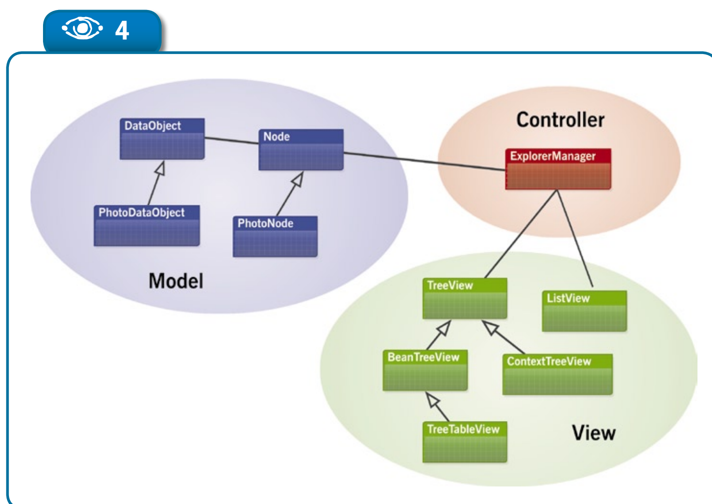


Figure 4 NetBeans MVC components.

The Lookup API also promotes decoupling

The default **Lookup** instance also contains the current set of selected **Node** objects. This makes it possible to design an effective and loosely-coupled mechanism also for inter-module communication. It's based on the Observer pattern: some modules publish their node selection to the default **Lookup**, while others listen for changes. And by implementing some filtering related to the kind of information associated to the changed nodes we get to the Publish/Subscribe design pattern.

For example, in blueMarine there are many ways to navigate the photo database and show a set of thumbnails – by exploring folders, the calendar, photos that share a tag, photos in the same gallery, and so on. The "explorer" modules just publish a selection of **Node**s bound to **PhotoDataObject**s to the default **Lookup**; a Thumbnail Viewer receives notifications and updates itself appropriately (see **Figure 5**).

The explorer components do not depend on the Thumbnail Viewer. Actually they are completely decoupled from it (we're applying Inversion of Control here). With this design I can add as many explorers as I want, even in independent modules that can be installed as add-ons. I can also easily add new viewers. For instance, I was able to include a Filmstrip Viewer as a completely decoupled component that can be used by itself or together with the original Thumbnail Viewer (see **Figures 6** and **7**).

The Lookup API has many other uses, as many types of objects (including **Node**s themselves) have their own local **Lookup** instance. I've only shown the "tip of the iceberg" here.

Listing 1. A Locator that uses the Lookup class.

```
public class CatalogLocator {
  public static final synchronized Catalog findCatalog() {

    final Lookup lookup = Lookup.getDefault();
    final Catalog catalog =
         (Catalog)lookup.lookup(Catalog.class);

    if (catalog == null) {
      throw new RuntimeException(
          "Cannot lookup Catalog");
    }
    return catalog;
  }
}
```

Listing 2. Retrieving registered objects.

```
private DefaultComboBoxModel mapProvidersModel =
    new DefaultComboBoxModel();

private void searchMapProviders() {
  Result result = Lookup.getDefault().lookup(
      new Template(MapProvider.class));

  for (Object provider : result.allInstances()) {
    mapProvidersModel.addElement(provider);
  }
}
```

The FileSystem API

Java offers just a bare-bones approach for file management through the **java.io.File** class, which wraps a file name and provides attribute access, basic operations and directory listing. This approach is really poor, since there's no concept of a filesystem; furthermore, **File** objects are bound to local, physical files/directories. What if you need to represent a virtual or remote directory tree? I faced this problem a few years ago, and in the end I solved it by extensively subclassing **File** – not a neat design, even though it worked.

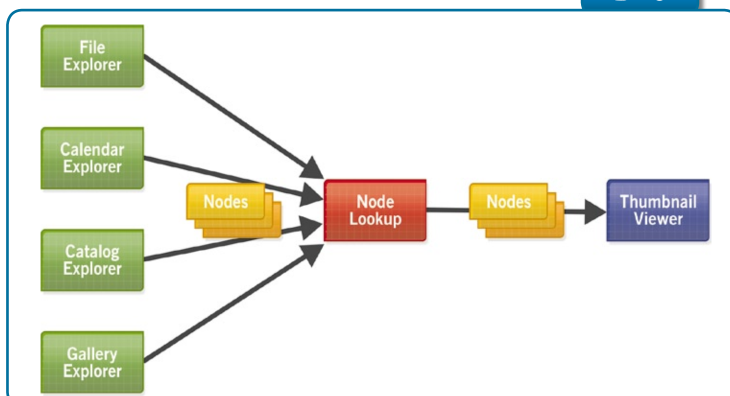


Figure 5 The role of *Node* objects for inter-module communication.

jrawio.dev.java.net The *jrawio* Image I/O plugin for RAW file formats

swinglabs.org SwingLabs

Figure 6 The Thumbnail Viewer and the Film Strip Viewer must show the same nodes – with the same selection too.

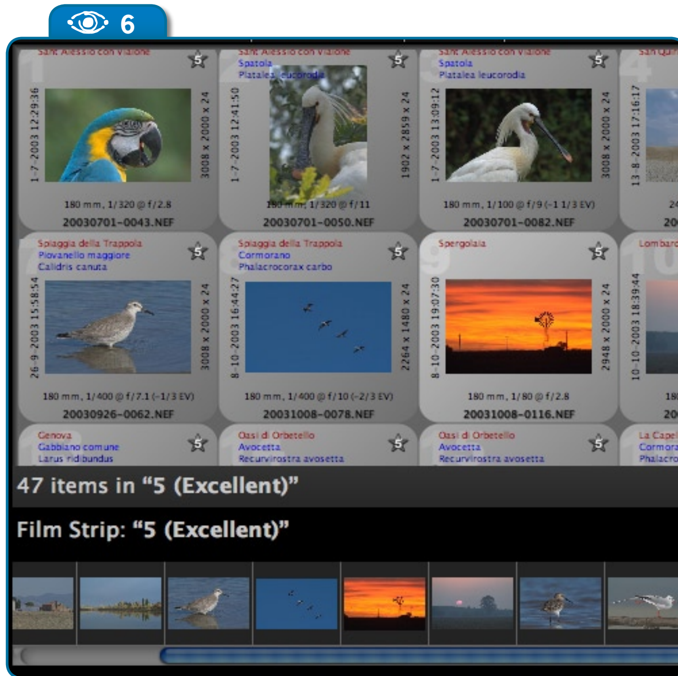
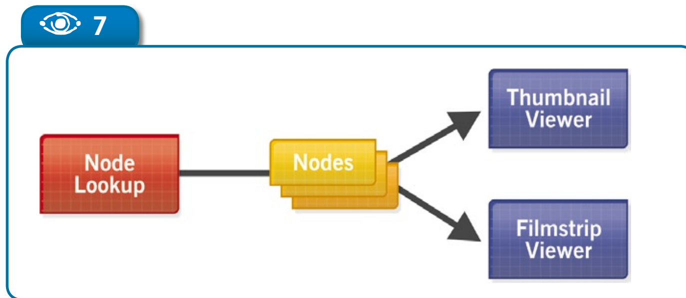


Figure 7 Multiple, synchronized views are implemented by just listening to the same *Node*'s changes.



NetBeans' **FileSystem** API fills this gap. There's a whole set of **FileSystem** classes that can be used to represent different types of filesystems: local, remote or even virtual. (NetBeans' inner configuration settings, including those of your custom code, are stored in such a virtual file system.) The only caveat is that you must use a NetBeans-specific instance of **FileObject** instead of **File**, but converting one into the other is easy:

```
FileObject fileObject = ...;
File file = ...;
file = FileUtil.toFile(fileObject);
fileObject = FileUtil.toFileObject(file);
```

blueMarine has a strict requirement for managing files. Each file must be associated with a unique id stored in a local database. The id is later used to build relationships between each photo and other entities such as thumbnails, metadata, galleries, editing settings, and so on. It's a rather common design for this kind of application, allowing you to later move or rename photos without too many changes in the database. It also lets you work with remote volumes such as external disks and DVDs. In other words, even when you don't have the file available in the system, you can look at its thumbnails and metadata.

A good starting point for tweaking the file-system management is the **LocalFileSystem** class, which represents a tree of files with a single root (for systems with multi-root hierarchies like Windows you just need to put a few **LocalFileSystem** objects together).

The **LocalFileSystem** class includes **AbstractFileSystem.Attr** and **AbstractFileSystem.List**. **Attr** lets you manipulate a set of attributes for each **FileObject**, and **List** lets you customize a directory listing. (Attributes are just simple properties which are bound to a **FileObject** and can be manipulated with getters and setters.)

I started by writing a simple subclass of **LocalFileSystem** that installs decorators of **Attr** and **List**, as shown in **Listing 3**. The **AttrDecorator** class retrieves the unique id for each file path (the **FileIndexer** is just a sort of DAO for this data), and makes it available as a special attribute of **FileObject** (**ID_ATTRIBUTE**). The code is shown in **Listing 4**.

While **AttrDecorator** would be enough to satisfy the functional specs, there's still the problem of batch loading. The **readAttribute()** method would be called quite randomly, thus preventing any effective batch policy (**FileIndexer** is able to do batching by lazy loading, but to be effective it needs to have a good number of entries to batch!).

Here **ListDecorator** helps us, as it intercepts children files after they are listed from a parent directory (see **Listing 5**). Calling **createIndexing()** immediately on the set

Listing 3. Plugging decorators into the LocalFileSystem class.

```
class LocalIndexedFileSystem extends LocalFileSystem {
  public LocalIndexedFileSystem() {
    attr = new AttrDecorator(attr, this);
    list = new ListDecorator(list, this);
  }
}
```

Listing 4. Retrieving registered objects.

```
class AttrDecorator implements AbstractFileSystem.Attr {
  private static final FileIndexer fileIndexer =
      FileIndexerLocator.findFileIndexer();

  private AbstractFileSystem.Attr peer;
  private LocalIndexedFileSystem fileSystem;

  public AttrDecorator(AbstractFileSystem.Attr peer,
      LocalIndexedFileSystem fileSystem)
  {
    this.peer = peer;
    this.fileSystem = fileSystem;
  }

  public Object readAttribute (String path, String name) {
    if (IndexedFileSystem.ID_ATTRIBUTE.equals(name)) {
      String path2 = fileSystem.findCompletePath(path);
      Serializable id = fileIndexer.findIdByPath(path2);

      if (id == null) {
        fileIndexer.createIndexing(path2, false);
        id = fileIndexer.findIdByPath(path2);
      }
      return id;
    }
    else {
      return peer.readAttribute(path, name);
    }
  }
  ...
}
```

Listing 5. Decorating directory scanning.

```
class ListDecorator implements AbstractFileSystem.List {
  private static final FileIndexer fileIndexer =
      FileIndexerLocator.findFileIndexer();
  private AbstractFileSystem.List peer;
  private LocalIndexedFileSystem fileSystem;

  public ListDecorator (AbstractFileSystem.List peer,
      LocalIndexedFileSystem fileSystem)
  {
    this.peer = peer;
    this.fileSystem = fileSystem;
  }

  public String[] children (String path) {
    String[] result = peer.children(path);

    if (!fileSystem.disableChildrenIndexing) {
      String path2 = fileSystem.findCompletePath(path);
      if (result != null) {
        for (String child : result) {
          fileIndexer.createIndexing(path2 + child, true);
        }
      }
    }
    return result;
  }
}
```

of listed files allows the **FileIndexer** to batch the retrieval of their ids.

Actions and Menus

Actions and Menus (together with auxiliary components such as toolbars) are the main facilities for interacting with the user. Swing provides basic support for them, but you soon realize that this is not enough, especially if you are designing a modular application.

Menus are organized hierarchically and grouped according to intuitive criteria. So a pluggable module will need to place its own menu items in the proper places (for instance, under a global "Edit" or "View" item in the menu bar), and possibly respect some meaningful sequence (e.g. "menu items of module C should appear between those of modules A and B"). Also you might like to introduce menu dividers to group some menu items together.

Swing actions can be enabled and disabled with a simple attribute change; but the implementation of the decision whether to enable or disable them is up to you. Often this is done by a special method that computes the states of a set of actions and is invoked after any change made by the user, as in:

```
private void setEnablementStatus() {
  myAction1.setEnabled(/* condition 1 */);
  myAction2.setEnabled(/* condition 2 */);
  ...
}
```

This approach works, but it's neither modular nor easily maintainable. And one must consider that in most cases the boolean conditions (condition 1, 2, etc. in the previous code) are just a

substance.dev.java.net The Substance look and feel

jai.dev.java.net JAI – Java Advanced Imaging

Look-and-feel examples with the NetBeans IDE

netbeans.org/competition/look-and-feel.html

function of the set of objects currently selected by the user – e.g., you can run "Edit" or "Print" only if a photo is selected.

Managing Menus and Actions with plain Swing in a clever way doesn't require rocket science, but you'll get a headache if it needs to be done from scratch for a sophisticated, modularized application. Fortunately, the NetBeans Platform also helps you in this area.

First, the Platform provides richer classes than Swing's **Action**. Some of the most commonly used are:

- **NodeAction** – A generic action that changes state when a new set of **Node**s is selected. The programmer must subclass it and override an **enable(Node[])** method, which evaluates the proper boolean expression for activating the action.
- **CookieAction** – This is an action whose state depends on the currently selected **Node**, and on it being bound to a given object (usually a specific **DataObject**). It also deals with different selection modes such as "exactly one", "at least one", etc.

After having implemented your action using the proper class, you declare it in your module's *layer.xml*, which acts as a generic configuration file (it models a "virtual file system" structured as the contained XML DOM).

Usually you don't have to do this manually: the NetBeans IDE offers a "New action" wizard that asks for the required information and both generates skeleton Java code and updates the relevant part of *layer.xml*. In fact, most of the XML in the following examples can be generated or manipulated through the IDE.

This approach works for both actions that should appear on contextual menus (see **Figure 8**) and for actions that need to be attached to a "regular" menu. In *layer.xml* you can also declare toolbars, where groups of buttons are logically bound to actions, and define keyboard shortcuts.

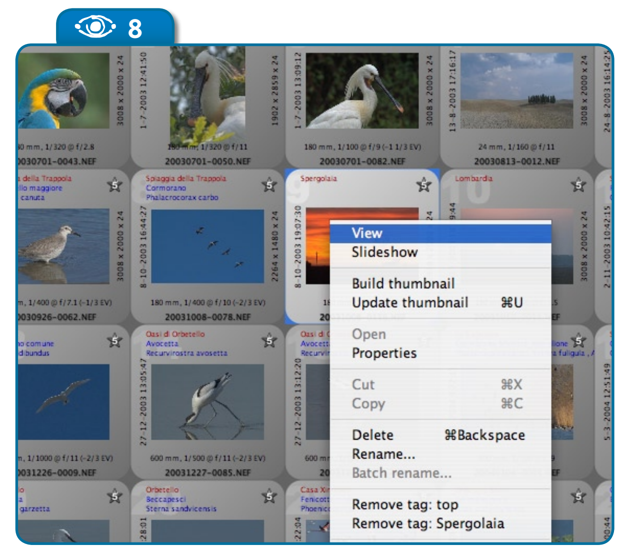


Figure 8 Each thumbnail is a rendered *Node* object bound to a *PhotoDataObject* and can pop up a menu with contextual actions. The grid arrangement and the look of thumbnails are implemented by means of a customized *ListView*.

The Windowing API

The NetBeans Platform provides a specific windowing component named **TopComponent.** This component models a rectangular portion of the main window, which can be resized and docked in different areas of the screen (these areas are called "modes"). For instance, the Explorer mode is a vertical column at the left side; the Properties mode is a vertical column at the right side; the Editor mode is the remaining space at the center (see **Figure 9**).

Also, a **TopComponent** can be activated or deactivated, has its own mechanism for keeping persistent state (which is automatically restored on restart), and can request attention by flashing its tab.

Docking areas can be resized with the mouse, or programmatically. You can assign **TopComponent**s to different areas by dragging them with the mouse or through code. You can define your own docking areas as well. For instance, I needed a component called "Film Strip" that should be placed at the bottom of the window. So I defined a docking area called "strip" and bound the Film Strip to it (see **Figure 9** again).

While this flexibility is great for some types of applications, such as an IDE or a CAD system, so much control could be distracting to some classes of users. For blueMarine I prefer not to have such flexible docking: a fixed scheme is used, offering just a bit of control through menu commands that let you swap the components in the Explorer and Properties modes.

The tabs and the control code which allows docking with the mouse has been removed in blueMarine by specifying a special **Tab-**

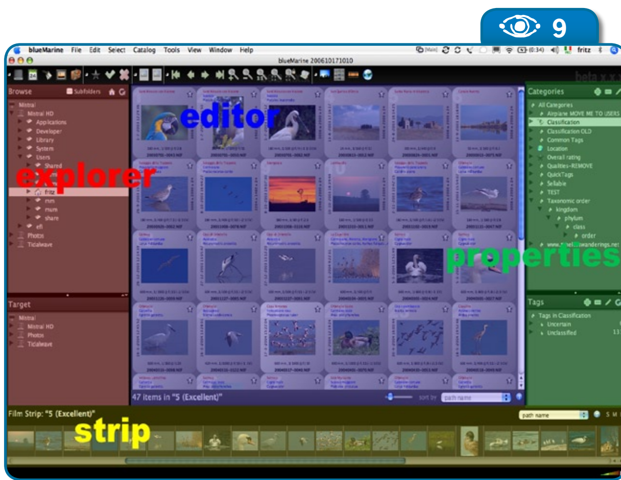


Figure 9
Modes in the main window of blueMarine – "strip" is a custom mode; each mode contains a *TopComponent*.

DisplayerUI (the visual component for each mode). I implemented programmatic control with the code in **Listing 6**.

Indeed it was not hard at all to implement the **TabDisplayerUI** trick, but only because I discovered the solution on Geertjan Wielenga's blog; without that help it would have taken much longer. I found that programmers can enjoy excellent support from the NetBeans community, both in the mailing list and in the evangelists' blogs – I recommend you to bookmark these!

The Look & Feel

The predefined Java look and feels are getting better with each JDK release, but sometimes you need a special LAF. For instance, dealing with photography you need a clean GUI that doesn't distract the user, and a darkish theme (where all the used colors are rigorously shades of gray), so as not to disturb a correct perception of colors.

Since JDK 1.4, the **UIManager** class allows you to plug different look and feels with minimal or no impact on existing code. As the class is a part of the standard Swing API, there are a lot of compatible LAFs that can be easily plugged into your app.

If you find a look and feel that you like, you can install it into NetBeans (and of course into your NetBeans Platform application) with a simple command-line switch:

```
--look-and-feel <name of the L&F class>
```

After some tests, I decided to keep the native look and feel for every piece of the GUI except for the main window, where I just changed the component colors (Mac OS X is a special case; see below). The typical blueMarine look and feel is illustrated in **Figure 10**.

As you know, changing the colors of a Swing component is usually a matter of **c.setForeground()** and **c.setBackground()**. Since the NetBeans Platform is Swing-based, things aren't much different. But there are a few exceptions. For instance, these standard methods don't work on **ListView** (one of the most used view components for **Node** objects). In blueMarine, this was fixed with the code in **Listing 7**, which first retrieves the inner **JList** and then changes its properties as needed. Similar code works with tree-based components (which share the same problem).

I found another problem with tree-based components: even with the code shown before, tree cells were rendered in black and white. Again, inspecting the sources, it was easy to find the cause: NetBeans' trees usually have a special cell renderer which does many things, such as supporting HTML rendering (so you can use multiple text styles); the cell renderer also chooses a color scheme that contrasts well between the foreground and the background. This is a clever idea in most cases, but not when you want to fine-tune your colors. The workaround was implemented by the few lines shown in **Listing 8**. Here's how you install the patched renderer:

```
PatchedNodeRenderer nodeRenderer = new PatchedNodeRenderer(tree.getCellRenderer());
tree.setCellRenderer(nodeRenderer);
```

weblogs.java.net/blog/joshy
Joshua Marinacci's blog

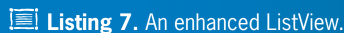
Listing 6. Programmatic component docking.

```
TopComponent topComponent = ...;
String newMode = "explorer";
Mode targetMode =
   WindowManager.getDefault().findMode(newMode);

if (targetMode != null) {
  component.close();
  targetMode.dockInto(component);
  component.open();
  component.requestVisible();
}
```

The author's blogs

The Quaqua look and feel


Listing 7. An enhanced ListView.

```
public class EnhancedListView extends ListView {
  protected JList jList;

  @Override
  protected JList createList() {
    jList = super.createList();
    jList.setOpaque(isOpaque());
    jList.setBackground(getBackground());
    jList.setForeground(getForeground());
    return jList;
  }

  @Override
  public void setBackground (Color color) {
    super.setBackground(color);
    if (jList != null) {
      jList.setBackground(color);
    }
  }

  @Override
  public void setForeground (Color color){
    super.setForeground(color);
    if (jList != null) {
      jList.setForeground(color);
    }
  }

  @Override
  public void setOpaque (boolean opaque){
    super.setOpaque(opaque);

    if (jList != null) {
      jList.setOpaque(opaque);
    }
  }
}
```

Regarding the look and feel, Mac OS X raised some particular issues. Mac users, who are really picky about aesthetics, noticed some time ago that even the Java implementation created by Apple does not accurately reproduce the operating system look and feel. This prompted the creation of a third-party product, named Quaqua, which fixes all the problems and implements a pixel-accurate Aqua GUI. (Actually the problems go beyond pixel accuracy: for instance the Java **JFileChooser** under Apple's Mac OS LAF is terrible in comparison to the native one.) As Quaqua is a regular look and feel handled by the **UIManager** class, its integration in blueMarine was not a problem, with the exception of a few Quaqua issues that were quickly fixed by the project's developer.

Listing 8. A patched cell renderer for controlling colors in JTree's.

```
class PatchedNodeRenderer extends DefaultTreeCellRenderer {
  private TreeCellRenderer peer;

  public PatchedNodeRenderer (final TreeCellRenderer peer) {
    this.peer = peer;
  }

  @Override
  public Component getTreeCellRendererComponent (
     final JTree jTree,
     final Object object, final boolean selected,
     final boolean expanded, final boolean leaf,
     final int row, final boolean hasFocus)
  {
    final Component component = peer.getTreeCellRendererComponent(
       jTree, object, selected, expanded, leaf, row, hasFocus);

     component.setBackground(selected ? getBackgroundSelectionColor() :
          getBackgroundNonSelectionColor());
     component.setForeground(selected ? getTextSelectionColor() :
          getTextNonSelectionColor());

    return component;
  }
}
```

Other extensions needed

A little more work was necessary with **ListView**, since the thumbnail-rendering components required a custom renderer (to provide the "slide" look and for overlaying extra information). Also needed was a grid layout with a predefined cell size. See **Figure 9**.

The standard **JList** makes these things quite easy to achieve. It's a matter of setting a few properties:

```
jList.setCellRenderer(...);
jList.setLayoutOrientation(JList.HORIZONTAL_WRAP);
jList.setFixedCellWidth(150);
jList.setFixedCellHeight(150);
```

But unfortunately NetBeans Platform developers left these methods out of **ListView** (much like the control of colors). With a similar workaround to the one used for the colors problem (i.e. accessing the inner **JList**), it was easy for me to add the missing

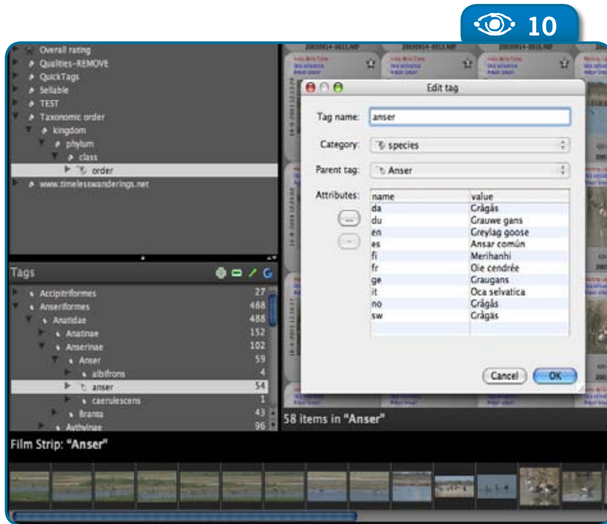


Figure 10 Using a dark color scheme in the main window and a regular scheme in popups.

methods to **EnhancedListView**, as you see in **Listing 9**.

As a final point, custom **ListCellRenderers** must add an extra step through a utility class, named **Visualizer**, to retrieve the **Node** associated to the current object to render, as shown in **Listing 10** (the usual **JList** approach, that is a cast applied to the **value** parameter, would not work).

Listing 9. New methods in ListView.

```
public class EnhancedListView extends ListView {
  ...
  public void setCellRenderer (ListCellRenderer listCellRenderer) {
    jList.setCellRenderer(listCellRenderer);
  }
  public void setFixedCellWidth (int size) {
    jList.setFixedCellWidth(size);
  }
  public void setFixedCellHeight (int size) {
    jList.setFixedCellHeight(size);
  }
  public void setLayoutOrientation (int layoutOrientation) {
    jList.setLayoutOrientation(layoutOrientation);
  }
}
```

Listing 10. Retrieving the Node object from a custom cell renderer.

```
import org.openide.explorer.view.Visualizer;

public class ThumbnailRenderer extends JComponent
    implements ListCellRenderer
{
  final public Component getListCellRendererComponent (
     JList list, Object value, int index,
     boolean isSelected, boolean cellHasFocus)
  {
    this.hasFocus = cellHasFocus;
    this.selected = isSelected;
    this.index = index;
    Node node = Visualizer.findNode(value);
    thumbnail = (Thumbnail)node.getLookup().lookup(Thumbnail.class);
    return this;
  }
...
```

Conclusions

Well, what can I say... blueMarine has survived its crisis, and its open-source “advertising” has already brought me a business opportunity for an application related to photo management (which reuses some of the back-end components of blueMarine – track my blogs if you want to know more about it). And without NetBeans, I would have dropped blueMarine and missed this opportunity.

What about blueMarine itself? When this article is published, the project should be very close to the first Release Candidate of its second life. Redesigning around the NetBeans Platform required some time, of course. It’s likely that you’ll need to deeply redesign an existing application if you want to take full advantage of all NetBeans Platform features (even though you could choose a lower-impact, more incremental way of working than I did). But I’m quite pleased with the results, and now I can add new features much faster than before.

To me, this means that Swing and the NetBeans Platform are now mature technologies for the desktop. I’m no longer forced to waste time reinventing the wheel. On the contrary: I can move very rapidly from an idea for my application, to an effective implementation.

Fabrizio Giudici (*fabrizio.giudici@tidalwave.it*) has a Ph.D. in Computer Engineering from the University of Genoa (1998), and begun his career as a freelance technical writer and speaker. He started up a consultancy company with two friends and since 2005 is running his own company. Fabrizio has been architect, designer and developer in many industrial projects, including a Jini-based real-time telemetry system for Formula One racing cars. He’s a member of the NetBeans Dream Team, the IEEE and JUG Milano.