Release 6.0 . JSF . Matisse . C/C++ . Mobility . NetBeans Platform . Scripting Languages

May . 2007

NetBeans magazine

Core NetBeans 6.0 Features
Know in depth what's coming in the new release

Introducing C/C++ Pack
Leverage NetBeans for native development

The blueMarine Project
NetBeans Platform development in the real world

OpenOffice.org Integration
Create add-ons and components to interface with OOo

Project Schliemann
Opening the IDE to other languages

Mobility Pack in Practice
Learn the basics and reduce device fragmentation

New UI Design Features
Upgrade your desktop productivity with NetBeans 6.0

Visual Web Development
Rapid web application design and implementation

Reach Out
with the IDE and Platform

NetBeans magazine
May . 2007

Publisher & Editor-in-Chief
Leonardo Galvão
leonardo.galvao@gmail.com

Assistant Editor
Osvaldo Doederlein
opinali@gmail.com

Design and Layout
pH Design

pH DESIGN
WWW.PHDESIGN.COM.BR

Graphic Designers
Tarcísio Bannwart
Jaime Peters Jr
Tersis Zonato

Illustrator
Felipe Machado

Contributors
Anatole Wilson
Beth Stearns
Fabrizio Giudici
Fernando Lozano
Geertjan Wielenga
Kay Koll
Osvaldo Doederlein
Wade Chandler

Community Support
Robert Demmer
Bruno Souza
Geertjan Wielenga

NetBeans Magazine is supported by NetBeans.org

Although every precaution has been taken in the preparation of this magazine, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information herein contained.

Reaching Out

Here we are at another NetBeans Day – the largest one in history. NetBeans 6.0 is coming, and the IDE and Platform teams are certainly keeping the rhythm, introducing new high-impact features and revamping traditional functionality at full throttle. All with the help of an expanding and vibrant community that reaches five continents.

The first anniversary issue of NetBeans Magazine goes along with the IDE's ever-widening scope. In eight in-depth articles you'll see what's upcoming in many areas and learn how to best use what's available today. We cover a lot of ground here: from native programming with C/C++ to scripting language support and OpenOffice.org integration – to mobile, Platform-based, and web development.

Strong competition in the Java ME arena is increasing demand for mobile productivity and tool support, and that's an area where NetBeans shines. You'll see a fast-moving introduction to the Mobility Pack, and learn how to develop for two very different devices while using the same codebase. Here the IDE helps reduce device fragmentation through the powerful project configurations feature.

The C/C++ Pack provides full native development support with IDE niceties like syntax highlighting, code completion and more. Our open-source software specialist takes you through all the necessary steps to get your environment ready for cross-platform development with the Pack. He also tackles a typical scenario demonstrating how to create a native library and integrate it with a Java application.

The Schliemann Project has been gathering a lot of attention lately, due to the power of expansion it will bring to NetBeans. With a simple but expressive new language, you'll be able to add editor support for almost any scripting language. See how this works in an article by a top NetBeans evangelist, giving a broad view that prepares you for what's to come in this young and promising project.

And what about the core Java IDE itself? We've got it covered with two articles that explore the new features in NetBeans 6.0 in detail. Through the eyes of a language and compiler specialist, you'll learn what is behind the new code editor features and the IDE's extended refactoring support (both internal and Jackpot-based). Another article examines the upcoming features in Release 6.0 for GUI development and related activities.

The NetBeans Platform is given plenty of space in this issue, with an article that delves into the NetBeans extension APIs and shows how they were used to build and extend a complex desktop application. The author writes about how his project benefited from a full redesign to make use of Platform features; he also highlights how the growing maturity of Swing components kept the project alive.

Additionally, we have a very practical article that demonstrates the main features available for visual web development with NetBeans, from the pen of an experienced technology writer who has coauthored several books in the Java Series. And you'll learn straight from the source about the brand new plugin module for building OpenOffice.org and StarOffice extensions with NetBeans and Java.

Happy coding!

Leonardo Galvao

Contents

04 **New Core NetBeans 6.0 Features in Depth**

Osvaldo Doederlein

From editing to browsing, to versioning, building and profiling, there is great news for everybody in the new NetBeans release.

16 **A New Handbook for NetBeans Platform Development**

Dig into the Platform with the help of three NetBeans veterans.

17 **Schliemann: Integrating Scripting Languages in NetBeans 6.0**

Geertjan Wielenga

See how a 19th century explorer called Heinrich Schliemann is inspiring the IDE to become fluent in many languages.

22 **User Interface Design in NetBeans 6.0**

Wade Chandler

Beans Binding and Swing Application Framework support, and features you've probably been dreaming about having in your IDE

30 **Write Once, Deploy Anywhere**

Anatole Wilson

Writing applications for mobile devices and reducing device fragmentation with NetBeans Mobility Pack

40 **Swing and the NetBeans Platform in the real world**

Fabrizio Giudici

See how the NetBeans Platform and advanced Swing components have made it easier to develop a complex desktop application and the lessons learned in building it.

52 **Introducing NetBeans C/C++ Pack**

Fernando Lozano

Prepare your environment for cross-platform C/C++ development with NetBeans, and put the C/C++ Pack to work creating a native library for Java applications

60 **OpenOffice.org NetBeans Integration**

Kay Koll

Using a new OpenOffice.org plugin module to create, build, deploy and debug OpenOffice.org/StarOffice extensions and client applications

72 **Visual Web Application Design with NetBeans**

Beth Stearns

Exploring some of the many exciting features for visual web application design and implementation in NetBeans

NetBeans 6.0

New Core Features in Depth

Osvaldo Doederlein

New features and improvements in the next release of NetBeans make it a better IDE for any kind of developer. From editing to browsing, versioning, building, debugging, profiling or visual design, there are great news for everybody.

It's that time again. A major, dot-zero release of NetBeans will be available soon – about a year and a half after 5.0, which introduced significant new features like the Matisse GUI builder, and extensive improvements in CVS integration, web services and module development, to cite but a few. In contrast, version 5.5 focused outside the core IDE by supporting several new Packs that increased NetBeans' overall functionality to a level still unmatched by any other open-source IDE. Now, is NetBeans 6.0 worthy of the bump in the major version number? You bet it is, and in this article we'll look at some of the most important and interesting new features in the core IDE.

Javac-powered

Let's begin by looking not at an end-user feature but at a core IDE technology that provides the foundation for many enhancements. Past releases of NetBeans, like many other programming tools, contained custom code to parse Java sources and assist in code understanding and manipulation tasks (like refactorings, hints and fixes, outlining, etc). The result was sometimes limited functionality: simple highlighting, non-bulletproof refactorings, and the lack of support for features like code completion everywhere Java code appears.

The obvious solution would be reusing the mature technology of the javac compiler to do all Java source processing. But javac was not designed to support the requirements of a modern IDE: it was written and tuned for batch execution, and to accept as input full compilation units, perform a complete compilation and produce *.class* files as output. IDEs have very different requirements, among which the most critical is working in memory only. Suppose that after each character you type, the IDE wants to analyze the entire class again so it can update syntax error indications, perform highlighting, and provide other features that depend on the code structure. One option would be to write the editor's current content to a temporary file, invoke javac and parse the resulting *.class* files. But this would be very inefficient.

A much better solution is to call javac in the same process (as a local library), then pass the current sources as an in-memory parameter and receive in return the data structures containing the same information that would be present in the class files (which wouldn't need to be created). Up to Java SE 5, this solution would be possible, but only using the proprietary – and often unstable – internal APIs of a Java compiler.

This situation changed with Java SE 6, which introduced JSR 199 (Java Compiler API) and JSR 269 (Pluggable Annotation Processing API). The Java Compiler API enables tight and efficient integration with javac (and other Java source compilers), and JSR 269 – although initially designed for annotation processing – provides a source-level equivalent of reflection metadata. Working together, these new APIs allow IDEs and other tools to dig deeply into the structural information that javac extracts from source code. Additionally, javac's implementation was enhanced and tuned for embedded and interactive use.

NetBeans was heavily updated to integrate with these new capabilities, enabling many improvements in the IDE (discussed below). The changes also promise future benefits: when Java SE 7 comes out with a new set of language enhancements, you should expect NetBeans' toolset to catch up very fast.

A new editor

Common sense says no product can be perfect in everything it does, but NetBeans is getting closer each day. Historically, NetBeans users have been proud of the IDE's complete coverage of Java platforms from ME to EE, its support for effective GUI building, and its intuitive UI and open architecture. On the other hand, the IDE lagged in certain areas, like in the code editor or refactoring. This could put off programmers very focused in source code... types who'll pick *emacs* over visual designers any day. Well, these problems are no more with NetBeans 6.0.

jackpot.netbeans.org

The Jackpot project.

www.netbeans.info/downloads/dev.php

NetBeans 6.0 development builds.

AST-based selection

Selecting words or lines is good enough for text editors, but when working with sources you often need to work with ranges of text that form coherent pieces of code. Say you want to copy all the code inside a **for** loop body[1] in order to paste it in another loop with similar logic. Just place the cursor in any blank position inside the loop body, press *Alt+Shift+Up* and you're done. The editor selects the innermost range of text that includes the cursor position, and delimits a node of the source's Abstract Syntax Tree.

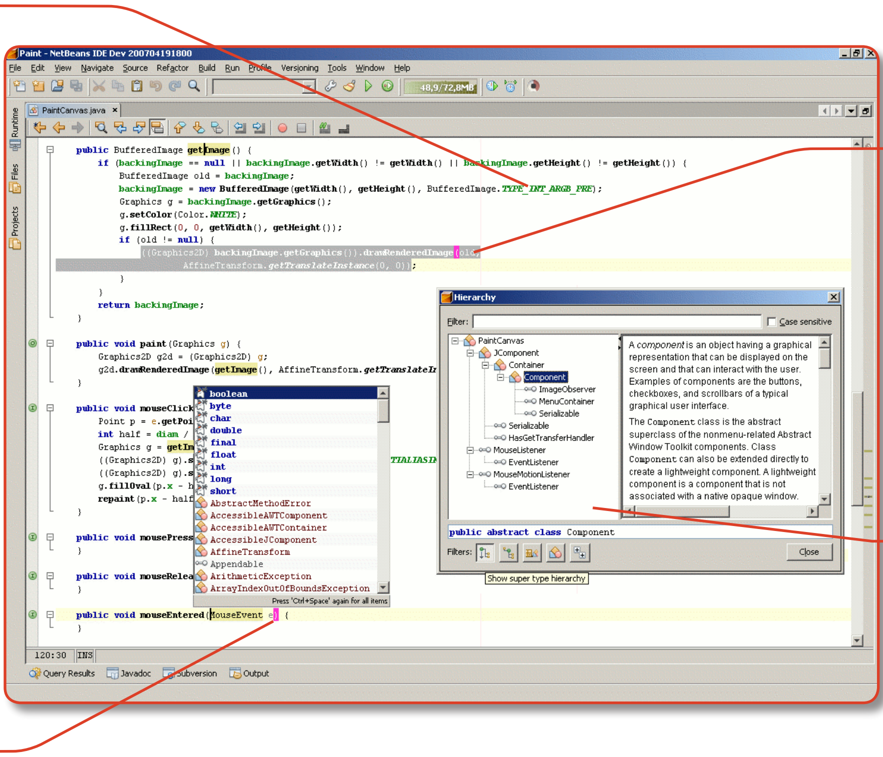
The Java compiler (as do most compilers) parses source code into an intermediary representation, which is structured as a tree. Each node in this data structure (called an Abstract Syntax Tree) represents a code element: a class, method, statement, block, identifier, operator, literal, etc. Though code processing tools usually manipulate programs as ASTs, many use a simple parser that produces only a basic tree. The "full" AST produced by a complete compiler like javac, which is capable of semantic analysis and code generation, will contain very detailed and reliable information about each node. For example, the node for an identifier holds not only its name but also its type and its "definite assignment" status (whether the identifier is guaranteed to be initialized at a given point); it can even hold its statically-calculated value (when applicable). Tools that work on top of a full AST are much more powerful and reliable. The difference won't be noticeable for a simple selection feature, but it may be very significant for more sophisticated functionality like refactorings.

Pressing *Alt+Shift+Up* again expands the selection to the next outer node, in this case the complete **for** statement; then a new keystroke may select the entire method, and so forth. *Alt+Shift+Down* will retract the selection to an inner node. **Figure 1** shows this feature being used to select a multi-line statement easily and precisely. I bet you will quickly be hooked on this feature and forget about all the other selection shortcuts! There's nothing like a code editor that groks *code*, not text.

Semantic highlighter

The editor's syntax highlighter was promoted to a semantics-aware highlighter. It can apply styles based not only on the types of tokens (like identifiers, operators or comments), but also based on different meanings that akin tokens may have – for instance, an identifier may be a class name or a local variable name, a parameter, a constant field, etc.

Figure 1
Several new editor features in action.


Paint - NetBeans IDE Dev 200704191800
File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help
48,9/72,8MB
PaintCanvas.java
Runtime
Files
Projects
Hierarchy
Filter:
Case sensitive
Filters:
Close
Show super type hierarchy
Press 'Ctrl+Space' again for all items
120:30
INS
Query Results
Javadoc
Subversion
Output

```
    public BufferedImage getImage() {
        if (backingImage == null || backingImage.getWidth() != getWidth() || backingImage.getHeight() != getHeight()) {
            BufferedImage old = backingImage;
            backingImage = new BufferedImage(getWidth(), getHeight(), BufferedImage.TYPE_INT_ARGB_PRE);
            Graphics g = backingImage.getGraphics();
            g.setColor(Color.WHITE);
            g.fillRect(0, 0, getWidth(), getHeight());
            if (old != null) {
                ((Graphics2D) backingImage.getGraphics()).drawRenderedImage(old,
                        AffineTransform.getTranslateInstance(0, 0));
            }
        }
        return backingImage;
    }

    public void paint(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.drawRenderedImage(getImage(), AffineTransform.getTranslateIr
    }

    public void mouseClick
        Point p = e.getPoi
        int half = diam /
        Graphics g = getIm
        ((Graphics2D) g).s                                    TIALIASIN
        ((Graphics2D) g).s
        g.fillOval(p.x - h
        repaint(p.x - half
    }

    public void mousePress
    }

    public void mouseRelea
    }

    public void mouseEntered(MouseEvent e) {
    }
```

```
boolean
byte
char
double
final
float
int
long
short
AbstractMethodError
AccessibleAWTComponent
AccessibleAWTContainer
AccessibleJComponent
AffineTransform
Appendable
ArithmeticException
ArrayIndexOutOfBoundsException
```

```
PaintCanvas
  JComponent
    Container
      Component
        ImageObserver
        MenuContainer
        Serializable
    Serializable
    HasGetTransferHandler
  MouseListener
    EventListener
  MouseMotionListener
    EventListener
```

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

The Component class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components. Class Component can also be extended directly to create a lightweight component. A lightweight component is a component that is not associated with a native opaque window.

```
public abstract class Component
```

Semantic highlighting (e.g., identifying usages of getImage(), and static variables in italics)

AST-based selection

Hierarchy view (opened on PaintCanvas)

Keyword completion at a method's parameter list.

[1] Depending on your bracing style, this may not be as easy as selecting a few full lines. There are many other examples, like selecting a complex expression that spans multiple lines.

One benefit of semantic highlighting is that it helps you take extra care when assigning to **static** fields (since many thread-safety and memory-leak bugs involve statics). **Figure 1** shows this off; notice that static fields (and references to these) appear in italics.

There are other powerful uses for the new highlighting engine:

- *Identifying usages* – Select any identifier, and the editor highlights all its uses in the same compilation unit. Again, **Figure 1** exemplifies this: clicking on a method name, all invocations to it are highlighted.
- *Flagging "Smelly code"* – The new editor highlights unused variables and imports, as well as usage of deprecated classes and methods. You don't need to perform a build or run a code lint tool to detect these simple (but frequent) problems anymore.
- *Exit and throw points* – Selecting a method's return type will highlight all **return** statements. Selecting an exception in the method's **throws** list will flag all **throw**'s of that exception type. All invocations to other methods that may throw the same exception are also flagged.

Better code completion

The bewildering amount of APIs you have to use these days makes code completion one of the most critical features of any modern code editor. NetBeans 6.0 has learned many new tricks here:

- *Keyword completion* – If you've just typed a **package** declaration in a new source file (for example), *Alt+Space* will bring only the keywords that are legal in that position: **abstract**, **class**, **enum**, **final**, **import**, **interface** and **public**. **Figure 1** shows another example: after the opening parenthesis of a method declaration, the preferred completions are all primitive types.
- *Type-based variable names* – Completing at "**ConfigurationFile** _", the editor will offer the variable names **cf**, **configurationFile** and **file**. (I'm using "_" to represent the cursor position.)
- *Generics-aware completions* – When assigning a variable with a generic type to a **new** expression, the editor will offer all compatible types, including generic arguments. For example, at "**Map<String, Integer> m = new** _", code completion lists all implementations of **Map**, each with the same **<String, Integer>** parameters.
- *Annotation-aware completions* – When completing after "**@**", you'll be offered all the annotations that can be used in the given scope. And if the selected annotation requires parameters the editor will provide completions for these too.
- *Passing parameters* – At "**x = m(**_", the top completions will be values in scope that are compatible with **m()**'s first parameter. If the method's parameter names are available and there are variables with similar names in scope, this is used to sort the completions further. You'll also be offered full completions with the parameter list filled with those variables.
- *Common constructors* – When you invoke code completion with the cursor positioned between class members, you'll be offered to create a constructor without arguments and one that receives initial values for all fields (if these constructors don't already exist).
- *Catching exceptions* – Completion at "**catch (**_" will only offer exceptions that are thrown in the corresponding **try** block, but haven't been handled yet by previous **catch** blocks.

New browsing views

The editor introduces several new views for source code browsing. The *Members view* shows the members of a Java type together with their javadocs, making it easy to find a particular method, field or inner class. The *Hierarchy* view shows the inheritance tree of a Java type. **Figure 1** demonstrates this view; notice the filter buttons that let you toggle between supertypes or subtypes and between simple and fully qualified class names. You can also choose whether or not to show inner classes and interfaces.

The *Declaration* view summarizes the declaration of the selected Java element (type, method or field). Despite its name, this view also shows the inspected element's source code if it's available. The Declaration View is especially useful when invoking code still under development, not yet documented with javadoc. Finally, the *Javadoc* view shows the javadocs for the selected Java element.

Editable Diff and Inline Diff

The editor's improved architecture makes it easier for various features that handle source code to integrate editor functionality. This is noticeable in the new Diff (opened, for example, by selecting a source file and choosing *Subversion>Diff*). When it's showing a local file, the right pane is editable, providing the full set of editor features – semantic highlighting and code completion included.

The new Diff adds other interesting tricks, like one-click merging and word-level diff (if a single word is changed in a line, only that word is highlighted). Check out these improvements in **Figure 2**.

You can also enable an Inline Diff feature, which creates a Diff sidebar, highlighting updated sections of a versioned file. The sidebar lets you visualize or rollback changes, and open the full Diff view.

Javadoc hints

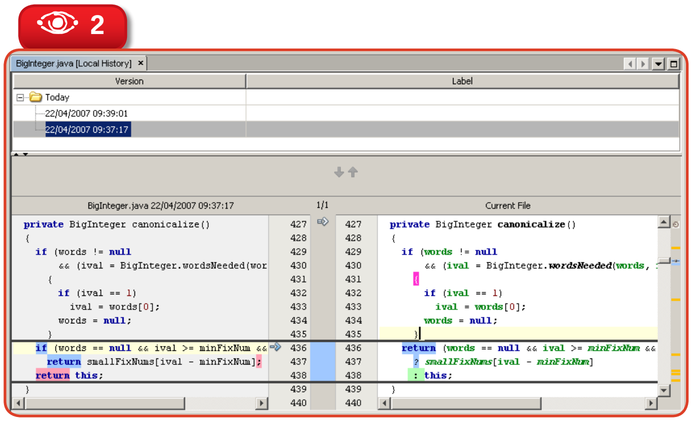
You always document all your code, right? Well, if you don't, NetBeans will complain about missing and incorrect javadoc tags. The IDE can help you with automatic fixes that add the missing tags, only asking you to fill in the blanks. And while you're doing that, you can use the new Javadoc view for convenient previewing.

Javadoc checking is active by default, but it's not intrusive: the editor will report missing javadoc tags just for the selected line; only incorrect tags will be reported everywhere. You can customize these and related options through *Tools|Options>Java Code>Hints*.

Other features

The new editor and its framework include other general features, like reusable editor tabs. These are useful for the debugger, to avoid cluttering your environment with editors opened by breakpoints or step-into's. There's also a new *Generate Code* dialog that automates the creation of constructors, getters and setters, **equals()** and **hashCode()**, and delegate methods.

2


BigInteger.java [Local History] ×
BigInteger.java 22/04/2007 09:37:17
1/1
Current File

Version	Label
Today	
22/04/2007 09:39:01	
22/04/2007 09:37:17	

```
private BigInteger canonicalize()
{
  if (words != null
      && (ival = BigInteger.wordsNeeded(wor
    {
      if (ival == 1)
        ival = words[0];
      words = null;
    }
  if (words == null && ival >= minFixNum &&
    return smallFixNums[ival - minFixNum];
  return this;
}
```

```
private BigInteger canonicalize()
{
  if (words != null
      && (ival = BigInteger.wordsNeeded(words, 
    {
      if (ival == 1)
        ival = words[0];
      words = null;
    }
  return (words == null && ival >= minFixNum &&
    ? smallFixNums[ival - minFixNum]
    : this;
}
```

Figure 2
The Local History and the new Diff: editing capability, semantic highlighting and word-level diff.

Refactoring and Jackpot

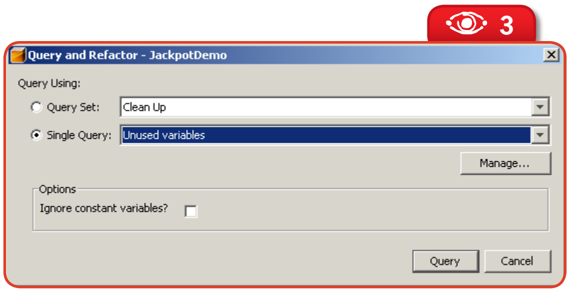
NetBeans 6.0 improves the existing refactoring support extensively. There is a new internal language-independent refactoring API that will allow implementing refactorings for code other than common *.java* sources (e.g., XML or JSF files). The new API also allows Java refactorings to precisely update dependent non-Java elements. This should make the current refactorings safer and easier to use.

The big news here, though, is the breakthrough new technology from project Jackpot, which has been available for some time but is only reaching maturity now. With its inclusion in NetBeans 6.0, Jackpot will be promoted to a standard feature and be more closely integrated with the IDE.

You may have heard that Jackpot is a new refactoring tool, but this really doesn't make it justice. Jackpot is actually a comprehensive framework for general code understanding and manipulation. You can use it as a replacement or foundation for several kinds of features: refactoring support, advanced searching and browsing, quality inspection, macro-like automation of complex editing tasks, and more.

Using Jackpot

Before taking a more in-depth look at Jackpot, let's show how easy it is to use. The new



Query and Refactor - JackpotDemo

Query Using:

○ Query Set: Clean Up

● Single Query: Unused variables

Manage...

Options

Ignore constant variables? □

Query Cancel

Figure 3 Jackpot's Query and Refactor dialog.

Query and Refactor command will show a dialog like **Figure 3**, where you can pick a Jackpot query or query set. Some queries have options that you can set to preferred values. Click *Query*, and any matches for the selected queries will appear in a view that details each match. Also, if the query involves code changes, you can preview and confirm these changes by clicking on a *Do Refactoring* button.

Jackpot rules

Jackpot's full power comes from its openness. This requires learning a new language but when you realize Jackpot's full potential you will see that the learning curve quickly pays off.

For example, here is a Jackpot query that detects an inefficient code pattern – the use of **equals("")** to check if a **String** is empty – and rewrites the matching code:

```
$s.equals(“”) => ($s.length() == 0) ::
    $s instanceof java.lang.String;
```

The syntax is **pattern => replacement :: condition**, where the **$** character identifies meta-variables that will bind to any Java program element (identifier, statement, operator, literal, etc.). Let's analyze each clause:

1. The pattern **$s.equals("")** matches invocations to the **equals()** method that pass an empty string as argument.

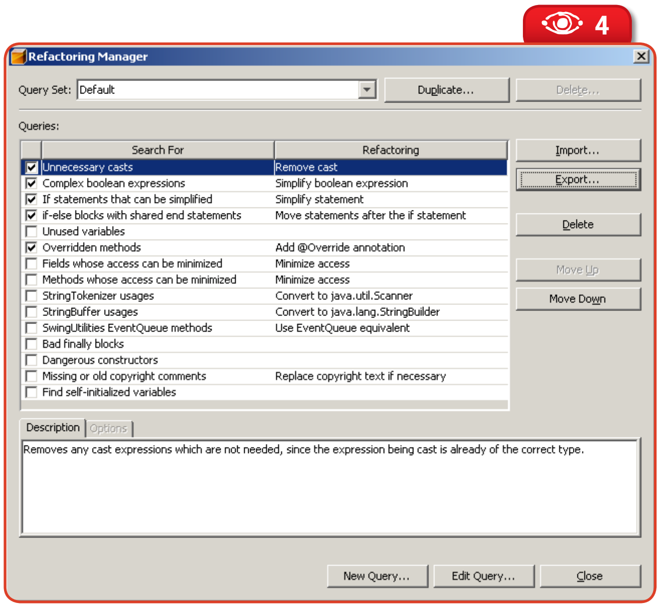
2. The condition is the only optional part of a rule in Jackpot's rule language, but it's a critical part in this particular rule: **$s instanceof java.lang.String** makes sure that the rule only fires when **$s** is a **String**. That's an important constraint, since our rule is specific to uses of **java.lang.String.equals()**, and not to just any implementation of **equals()**.

3. Finally, the replacement – **($s.length() == 0)** – rewrites the matching code.

There's a lot of sophistication behind this apparently simple behavior. For one thing, look at Jackpot's **instanceof** operator. It walks and quacks like Java's **instanceof**, but it's not the same thing. Java's **instanceof** is a runtime operator whose left-hand operand is an object reference. Jackpot's **instanceof,** however, is a compile-time (static) operator; its left-hand operand is any node of the program's AST.

Because Jackpot – like the new editor – relies on javac's source analysis engine, it's able to fully attribute all types in the processed code. This includes the most complex cases, like inferred generic types. Other code analysis tools often resort to heuristics that approximate types but might fail to calculate types for some expressions.

You could even try to do our refactoring (replacing **s.equals("")** by **s.length() == 0)** using plain regular expressions: search for **(\w*)\.equals\(\"\"\)** and replace it with **$1.length() == 0**. But regexes are rigid and dumb; they won't



Refactoring Manager

Query Set: Default Duplicate... Delete...

Queries:

	Search For	Refactoring
☑	Unnecessary casts	Remove cast
☑	Complex boolean expressions	Simplify boolean expression
☑	If statements that can be simplified	Simplify statement
☑	if-else blocks with shared end statements	Move statements after the if statement
□	Unused variables	
☑	Overridden methods	Add @Override annotation
□	Fields whose access can be minimized	Minimize access
□	Methods whose access can be minimized	Minimize access
□	StringTokenizer usages	Convert to java.util.Scanner
□	StringBuffer usages	Convert to java.lang.StringBuilder
□	SwingUtilities EventQueue methods	Use EventQueue equivalent
□	Bad finally blocks	
□	Dangerous constructors	
□	Missing or old copyright comments	Replace copyright text if necessary
□	Find self-initialized variables	

Import... Export... Delete Move Up Move Down

Description | Options

Removes any cast expressions which are not needed, since the expression being cast is already of the correct type.

New Query... Edit Query... Close

Figure 4 Jackpot's Refactoring Manager.

even exclude text that's inside comments or string literals, and a simple line break will prevent detection. This is obviously a straw man example (other tools, like PMD and FindBugs, are much smarter than regexes – although not up to javac-like precision), but it shows the value of smarter tools/features.

There are Jackpot operators without Java counterparts, from simple ones like **isTrue(*node*)**, which matches boolean expressions that can statically be proven to always evaluate to **true** – to more powerful operators like **isSideEffectFree(*node*).** The latter matches a statement, block or method that doesn't modify any variable outside its scope.

Again, such detections resemble existing code inspection tools, which detect problems like "dead code". But Jackpot's reliance on the full javac technology results in fewer false positives in detections, and higher safety in automatic replacements.

You can also write Jackpot queries in plain Java, using Jackpot APIs and NetBeans' module development features. This is necessary for complex rules that go beyond the capabilities of Jackpot's rule language. But as this language evolves, fewer and fewer queries should require implementation in Java. Performance, by the way, is not an issue: queries written in the Jackpot rule language are converted to Java and execute as compiled code.

Figure 4 shows Jackpot's *Refactoring Manager*. This configuration dialog allows you to inspect all installed queries and organize them into query sets. You can also import new queries. If you write a new query script, just click *Import* and the new query will be available in the *Query and Refactor* dialog.

Usage and perspectives

Jackpot ships with a library of predefined queries, containing many rules for code clean-up and detection of common programming mistakes or code anti-patterns, as well as migration of deprecated API usage.

As I write this, Jackpot has just been integrated into NetBeans. So we have a hybrid system with Jackpot co-existing with traditional refactoring and code manipulation features. This means that commands like *Rename method* are still implemented in the old-fashioned way, even though they could be implemented by a Jackpot rule. The same holds for code validations ("hints") and their automatic fixes. Some of this functionality will certainly be reimplemented on Jackpot in the future. Also, because Jackpot makes the development of such things much easier, you should expect an increasing number of refactorings, validations and other code-crunching features to be added to the IDE.

Extended Ant and JUnit support

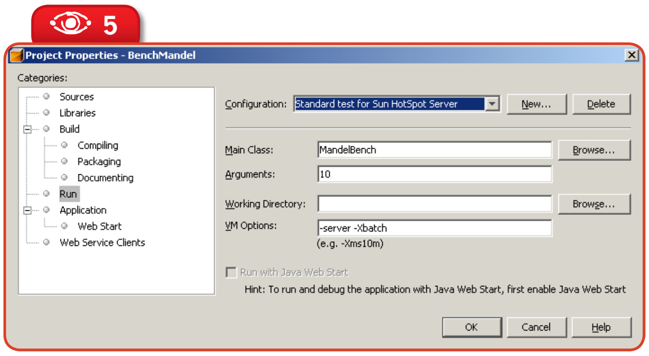
Ant support in NetBeans 6.0 has been updated to Ant 1.7.0, a major new release that adds such features as support for JSR 223-compatible scripting languages. There's also a new progress indicator for Ant processes.

The IDE's JUnit support now handles the annotation-driven test cases of JUnit 4. Old JUnit 3.8 test cases are still supported. Also, the project properties editor is improved with classpath entries specific to unit tests.

Project and build features

Editing code is fundamental, but for most non-trivial projects a well-structured and powerful build system is critical too. NetBeans' project management and build system was improved with many new features.

In addition to its Ant support, NetBeans can open and understand Apache Maven 2 projects. Though the new Maven-based proj-



Configuration: Standard test for Sun HotSpot Server

Main Class: MandelBench

Arguments: 10

Working Directory:

VM Options: -server -Xbatch
(e.g. -Xms10m)

□ Run with Java Web Start

Hint: To run and debug the application with Java Web Start, first enable Java Web Start

Figure 5 Multiple Configurations and support for Java Web Start in the new Project Properties dialog's Run page.

ect support is not intended to replace Ant projects anytime soon, it will be welcome to Maven fans or to anybody needing to build a project that requires Maven.

Also, now you can specify packages or classes to exclude from the source tree. This is useful for working with large projects, when you're not interested in seeing or running all of their code and a partial build is viable.

If you have many correlated projects, you can organize them into Project Groups, so certain operations like opening projects can be applied to the group as a whole. And if you write Java SE projects with many entry points (classes with **main()** methods), or with command-line parameters that require frequent edits of the project properties, the *Run Configurations* feature will make your life easier. The project properties' *Run* page shows a new *Configuration* option. Each configuration allows you to define the main class, arguments and VM options, independently of other configurations. See an example in **Figure 5**.

Furthermore, the new Java Web Start support automates the creation and maintenance of JNLP files, and makes it easier to run tests without needing a browser. In the Project Properties, check *Application>Web Start>Enable WebStart*, and off you go. Java Web Start support integrates with the Run Configurations feature, by creating a *Web Start* configuration. So you can test the same project with or without JAWS.

Version control

Robust version control is an essential feature, even for simple projects written by one developer over a weekend. For one thing, it's critical to enable "fearless programming", e.g. using techniques like refactoring (manual or automatic) without worry. NetBeans 6.0 brings plenty of news in this area too.

CVS

NetBeans has traditionally supported the CVS version control system and this support was already excellent in NetBeans 5.5. Version 6.0 adds several updates in usability, like exporting a diff patch of files selected in the Search view; a new command to open a specific revision, tag or branch; and an improved history search feature with new Summary and Diff views. There are also new advanced operations like changing the CVS root and doing a partial merge.

Subversion

The biggest news for many users, though, is support for the increasingly popular Subversion version control system. NetBeans 6.0 is the first release to integrate complete first-class support for SVN. Even though NetBeans 5.5 now offers a Subversion module in the Update Center, you really want version 6.0 if you are a heavy Subversion user.

Local History

No matter which Version Control System you prefer, you'll love the new Local History feature, already depicted in **Figure 2**. NetBeans 6.0 automatically keeps an internal history of recent changes to project resources. Every time you save a file, this is registered as a "commit" of a new version of the file in the local history. So file changes are tracked with fine granularity – somewhat like a persistent undo feature. You can inspect the "versions" in the local history and diff them against the current files.

Be warned, however, that this feature is mostly useful for undoing mistakes that escape the editor's undo capacity, e.g. after closing the editor or restarting the IDE. You can then revert to a previous state that you haven't yet committed to a safer VCS repository, perhaps because the new code was still rough and untested. The Local History feature is powerful and is sometimes a lifesaver, but it's not a full replacement for a real VCS.

Debugging

The debugger is of course among the most critical features of an IDE, and NetBeans is already very complete in this area. So what's left to improve in 6.0? First off, the Java SE 6 release contains two

nbi.netbeans.org

The new NetBeans Installer. As of this writing, you must follow a link to a directory where you'll navigate to the installer page for a specific build important new JVM debugging features which require an updated debugger to use. (The debuggers from NetBeans 5.5 or older releases won't benefit from these even if you run them on top of Java SE 6.) There are also other debugger improvements that are not dependent on the JRE version, so you'll benefit even if you are chained to some stone-age Java runtime like 5.0 or, heavens forbid, 1.4.2.

Forcing return values

Suppose you're stepping anywhere in a method and you'd like to force it to return immediately and produce a specific return value. This is now supported in the 6.0 debugger, letting you check "what-if" scenarios and reproduce bugs more easily. You won't need hacks like patching the source code with **return** statements (and having to unpatch it later). As I write, this feature is not yet implemented, but it should be before the final release.

Expression stepping

Expression stepping is another smart timesaver. In complex expressions containing method calls, you can step into individual invocations, and when such a call returns you can see the returned value even it's not assigned to any local variable. You no longer have to break expressions into simple parts and introduce temporary locals for the single purpose of helping debugging. Also, the Local Variables view will show the value returned by invoked methods.

Expression stepping will work in any Java runtime, but showing values returned by invoked methods requires Java SE 6.

Multithreading support

Another new feature that's very useful is *Debug current thread*: you can instruct the debugger so that only a given thread will stop in breakpoints. This is crucial for debugging concurrent applications that have several threads running the code of interest. Since *we* developers are not multithreaded, we're easily overwhelmed when setting a breakpoint causes the debugger to stop twenty threads at once!

Other features

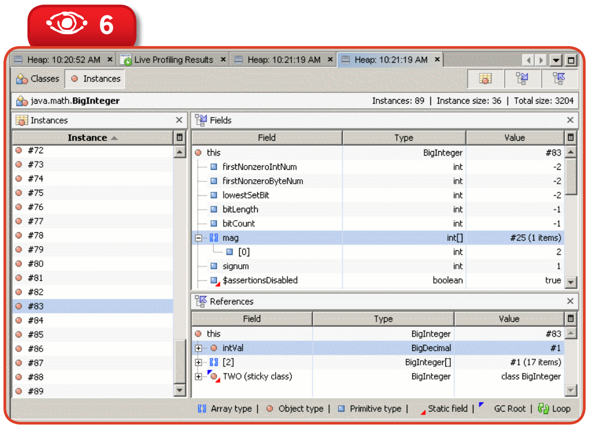
There are also general improvements to other features, like better handling of broken breakpoints (e.g. with incorrect conditions), and a command to copy call stacks to the clipboard.

New Profiler features

In NetBeans 6.0, the Profiler becomes part of the core distribution, and there's a range of important improvements.

- *Better performance* – Performance is good anywhere but it's always a critical issue in profilers. The NetBeans Profiler, which derives from Sun's JFluid research project, pioneered a new technology that allows profiling apps nearly at full speed by dynamically instrumenting code. Also, the Profiler itself should be fast to analyze and present data collected from the JVM – especially online data that's constantly updated as the program runs. The new release improves significantly the performance of the Live Results categorization and drill down, so you'll find yourself using this feature more often.
- *Classloading telemetry* – The VM Telemetry view now shows the number of loaded classes together with the number of threads.
- *Memory snapshot comparison* – Your application has a method that's suspect of leaking? Take heap snapshots before and after running it then diff the two snapshots.
- *Heap Walker* – The ultimate tool for leak hunting and any kind of memory allocation analysis. You can load a heap dump and visualize the full object graph in the heap (see **Figure 6**).

6



java.math.**BigInteger** Instances: 89 | Instance size: 36 | Total size: 3204

Instances

Instance
#72
#73
#74
#75
#76
#77
#78
#79
#80
#81
#82
#83
#84
#85
#86
#87
#88
#89

Fields

Field	Type	Value
this	BigInteger	#83
firstNonzeroIntNum	int	-2
firstNonzeroByteNum	int	-2
lowestSetBit	int	-2
bitLength	int	-1
bitCount	int	-1
mag	int[]	#25 (1 items)
[0]	int	2
signum	int	1
$assertionsDisabled	boolean	true

References

Field	Type	Value
this	BigInteger	#83
intVal	BigDecimal	#1
[2]	BigInteger[]	#1 (17 items)
TWO (sticky class)	BigInteger	class BigInteger

Array type | Object type | Primitive type | Static field | GC Root | Loop

Figure 6 The Profiler's Heap Walker, inspecting a particular instance of BigInteger.

- *Load generation* – The Profiler supports integration with load generation tools (currently only Apache JMeter is supported but more is to come).
- *Profiling Points* – These are a profiler's equivalent of debugger breakpoints. You can define places in your source code where the profiler should start/stop the clock, reset profiling results or take a snapshot. The Profiling Points feature removes most bureaucratic profiling work: never again will you need to step or pause code to get snapshots in critical events; you also won't need to tweak code to measure the latency of a region that doesn't coincide with a full method.

GUI and usability

An IDE should have a beautiful, efficient and productive GUI as much as any other application. NetBeans 6.0 makes new strides in this direction.

Linux and Solaris users will certainly welcome the much improved GTK L&F, which is now activated by default on these platforms. The activated-by-default part depends on Sun's JRE 6 Update 1 (or better), which contains its own share of important GTK updates. NetBeans will respect all settings from the active GTK theme.

The new NetBeans Installer (NBI) makes installation easier and faster. In the downloads page, you can select which packs you want (e.g. Enterprise, Mobility). Then you'll be offered a custom installer that includes all chosen features and will install these in a single go. NBI is especially convenient for system administrators that need to install the same IDE configuration in multiple machines, and for trainers who often land in unprepared laboratories.

NetBeans also includes redesigned icons, and the SDI windowing option (a relic from ancient NetBeans releases) was removed. Now you have undockable/floating windows. Finally, in the QA front, the new Report Exception tool streamlines reporting of detailed error data to NetBeans' developers, while the UI Gestures Collector can submit data about your IDE usage patterns. This data is useful not only for research, but also to implement a kind of "tip of the day" hint system not based on **Math.random().** I tested this, and the NetBeans Analytics site offered me a tutorial about profiling multithreaded programs, which was highly correlated with the tasks I had been performing in recent days.

Matisse and visual web development

There are only two core IDE features I'm not covering here. Both are award-winning tools and top reasons for many developers having moved to NetBeans: the Matisse visual editor, and the Visual Web Pack. NetBeans 6.0 brings significant updates to both. For Matisse, check out the article "UI Design in NetBeans 6.0" in this issue, where you'll find detailed information about what's new.

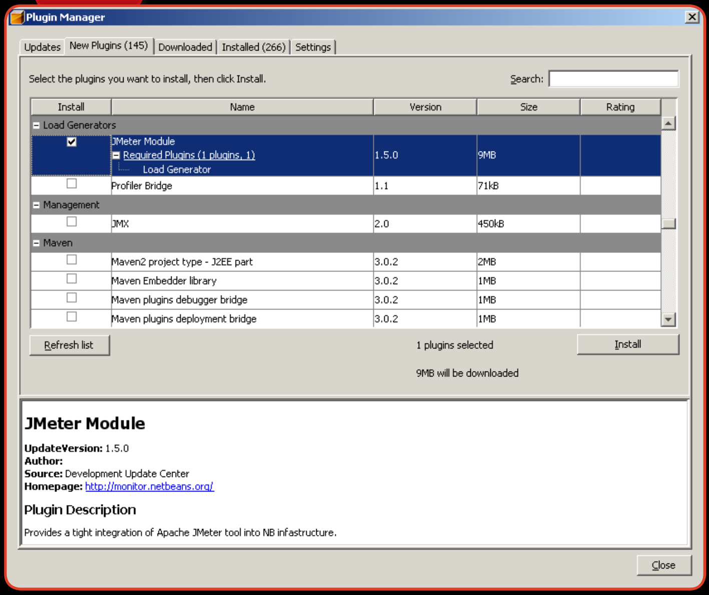
Currently, the most important changes in the Visual Web Pack refer to its integration into the NetBeans core. Actually, there won't be an external Web Pack for 6.0. The IDE already offered support for web application development, so it was a little odd to have some of that in the core and the rest in an external Pack. Historically, this happened because the Web Pack technology was originally developed as a separate product (Sun's Java Studio Creator), which was based on a fork of a very old NetBeans version. So its implementation became partially redundant with NetBeans' web tooling. Now this chasm is closed and there will be no more duplicate code or effort. The merge results in a simpler IDE for all users: from visual-design lovers to tag-writing diehards.

There are several new features in the integrated web tooling but as we write they are still under heavy development, so it wasn't viable to cover the new functionality in this issue. However, don't miss the article "Visual Web Application Design with NetBeans" for an updated tutorial on the last stable version.

Plugin Manager

NetBeans' open, extensible architecture is one of its core advantages and it's also very easy to use and integrate with. You may be surprised that the *Tools>Update Manager* has disappeared, though.

[2] Incidentally, several menu options were simplified in NetBeans 6.0; for instance, *Java Platform Manager* became *Java Platforms*.



Plugin Manager

Updates | New Plugins (145) | Downloaded | Installed (266) | Settings

Select the plugins you want to install, then click Install.

Search:

Install	Name	Version	Size	Rating
⊟ Load Generators				
☑	JMeter Module ⊟ Required Plugins (1 plugins, 1) Load Generator	1.5.0	9MB	
☐	Profiler Bridge	1.1	71kB	
⊟ Management				
☐	JMX	2.0	450kB	
⊟ Maven				
☐	Maven2 project type - J2EE part	3.0.2	2MB	
☐	Maven Embedder library	3.0.2	1MB	
☐	Maven plugins debugger bridge	3.0.2	1MB	
☐	Maven plugins deployment bridge	3.0.2	1MB	

Refresh list

1 plugins selected

Install

9MB will be downloaded

JMeter Module

UpdateVersion: 1.5.0
Author:
Source: Development Update Center
Homepage: http://monitor.netbeans.org/

Plugin Description

Provides a tight integration of Apache JMeter tool into NB infastructure.

Close

Figure 7 The new Plugin Manager.

But just look again, at *Tools>Plugins*[2], and you'll see **Figure 7**.

The new UI unifies and better organizes the old Update Center (see the *Updates*, *New Plugins*, *Downloaded* and *Settings* tabs), and also the old module manager (see the *Installed* tab). There are new features too: for example, when you select a plugin (like we did for the JMeter Module in **Figure 7**), a *Required Plugins* node will appear if applicable; you can expand it to see any dependencies that must also be installed.

Conclusions

NetBeans 6.0 comes with a massive number of new and improved features and certainly deserves the major version bump. If NetBeans 5.5 was wide, NetBeans 6.0 is also deep. Developers upgrading to the latest version will have not only extensive support for all kinds of Java development but also a best-of-breed feature set in every important functionality area.

Many NetBeans power users may have gone through this article and found features that were already available for previous versions via additional modules. From several editor enhancements to Run Configurations, to the Local History, you could find an *nbm* file that would provide some level of support for your need. However, you can now just install the core IDE and have all these features out of the box – and they're superior, more polished and better integrated than what's provided through external modules. This happens of course with every new release, but NetBeans 6.0 makes a very noticeable effort to catch up with its RFEs, embracing a large number of improvements that first surfaced as contributions from the broader community. This can only be viewed as great news, and as evidence of a project that moves fast in the direction users want.

Osvaldo Pinali Doederlein (*opinali@gmail.com*) is a software engineer and consultant, working with Java since 1.0 beta. He's an independent expert for the JCP, having served for JSR-175 (Java SE 5), and is a Technology Architect at Visionnaire Informatica. Osvaldo has an MSc in Object Oriented Software Engineering, is a contributing editor for Java Magazine and maintains a blog at *weblogs.java.net/blog/opinali*.

pH DESIGN

1 Year

NetBeans magazine

With this third issue, NetBeans Magazine is completing its first anniversary. Kudos and thanks to the NetBeans developer community for enabling us to spread the word even more about this wonderful IDE and Platform!

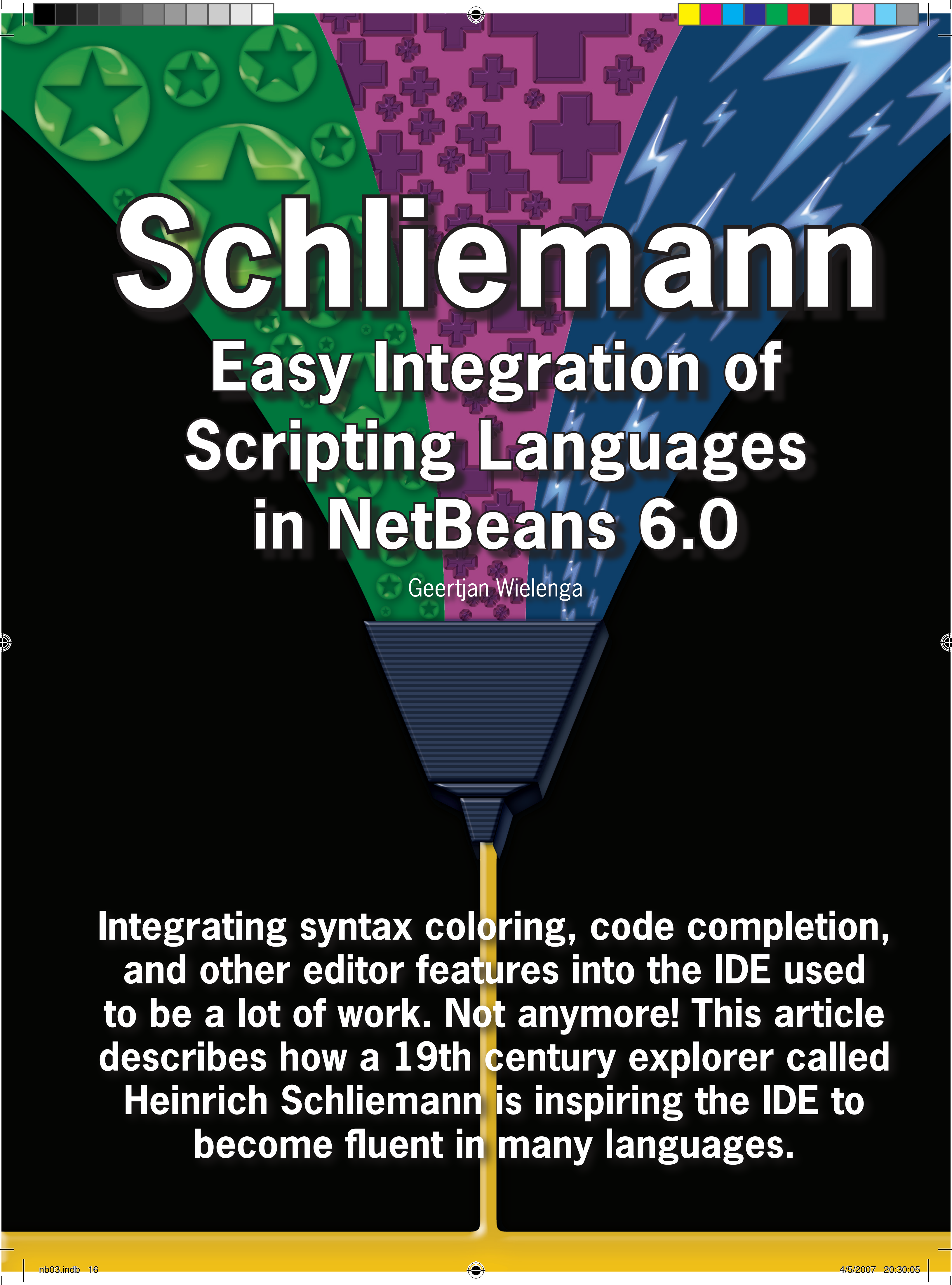
Release 6.0 . JSF . Matisse . C/C++ . Mobility . NetBeans Platform . Scripting Languages
May . 2007
NetBeans
magazine
Core NetBeans 6.0 Features
Know in depth what's coming in the new release
Introducing C/C++ Pack
Leverage NetBeans for native development
The blueMarine Project
NetBeans Platform development in the real world
OpenOffice.org Integration
Create add-ons and components to interface with OOo
Project Schliemann
Opening the IDE to other languages
Mobility Pack in Practice
Learn the basics and reduce device fragmentation
New UI Design Features
Upgrade your desktop productivity with NetBeans 6.0
Visual Web Development
Rapid web application design and implementation
Reach Out
with the IDE and Platform

netbeans.org/community/magazine

Schliemann

Easy Integration of Scripting Languages in NetBeans 6.0

Geertjan Wielenga



Integrating syntax coloring, code completion, and other editor features into the IDE used to be a lot of work. Not anymore! This article describes how a 19th century explorer called Heinrich Schliemann is inspiring the IDE to become fluent in many languages.

Traditionally, when creating editor support for a new programming language in the IDE, a vast variety of NetBeans APIs must be implemented. By "editor support", we typically mean syntax coloring, code completion, and the source navigation features provided by the IDE's Navigator. Other examples include code indentation and brace matching. Out of the box, the NetBeans IDE provides this kind of support for several languages and technologies, such as Java (of course), JSP, and HTML.

There are many NetBeans APIs that one needs to implement to provide editor support for a programming language. This is unfortunate for two reasons. Firstly, the domain knowledge that a language programmer typically brings to the table is the language itself, not the versatile knowledge of the NetBeans APIs required to provide the necessary features. Secondly, the underlying infrastructure for editor support is the same for all languages. For example, the only difference between the Navigator for Java and the Navigator for HTML is the actual code, not the container. For these reasons, the language programmer should only need to provide the *content* of the language in the form of *tokens* that are communicated in regular expressions. Nothing more than that should be needed.

Given the tokens and an indication of where they should be used, the NetBeans Platform should be able to figure out how to hook the tokens to the support features. Not only would this approach simplify the process of integrating a new language into the IDE, but it would leverage the *current knowledge* of the language programmer – rather than requiring a steep learning curve of acquiring new knowledge before coding can even begin.

Enter Schliemann

This, in sum, is what the new Schliemann project (*languages.netbeans.org*) is all about. And why is it called Schliemann? Heinrich Schliemann was a 19th century explorer who had a gift for languages. He traveled the world while keeping a diary in the language of the country he happened to be in. In the spirit of Schliemann, the 6.0 release of the NetBeans Platform envisages the IDE as being Schliemannesque, able to pick up languages very quickly and then being able to communicate in them fluently.

The project is especially pitched towards *scripting* languages, because the Schliemann project does not provide compilation support, which is not required by scripting languages – and because scripting languages, in particular, are increasingly in vogue today. In this article, we will explore the main facets of the Schliemann project and touch on some contrasts with the traditional NetBeans API approach to providing the editor features it supports.

Everything in a single file!

A central contrast between the traditional API approach and the Schliemann approach is that the latter lets you specify all editor features declaratively in one single file. This file has the *.NBS* file extension, which stands for NetBeans Scripting. To get a quick flavor of some typical content of an NBS file, let's examine a code snippet – see **Listing 1**.

This template is what you are given when you use the new Generic Languages Framework wizard, which is part of NetBeans IDE 6.0. It gives you a single NBS file with sample content, which begins with the definition of four tokens. These tokens are named "keyword", "operator", "identifier" and "whitespace". Within brackets, in the same line as the name of the tokens, a regular expression is used to define them.

Right away, one can see the power of this new approach to language support provision: a regular expression language, rather than Java, is used to define tokens. As a result, programmers outside the Java ecosystem can integrate their programming languages into the NetBeans IDE. Not needing to know Java, at least for the simpler integrations of languages, is a central benefit of the Schliemann

languages.netbeans.org

The official Schliemann project page on netbeans.org

wiki.netbeans.org/wiki/view/Schliemann

The Schliemann page in the NetBeans Wiki

Listing 1. NBS file snippet.

```
#
# NBS Template
#

# definition of tokens
TOKEN:keyword:( “while” | “if” | “else”)
TOKEN:operator:( “{“ | “}” | “(“ | “)” )
TOKEN:identifier:( [“a”-”z”] [“a”-”z” “0”-”9”]* )
TOKEN:whitespace:( [“ “ “\t” “\n” “\r”]+ )

# parser should ignore whitespaces
SKIP:whitespace

# definition of grammar
S = (Statement)*;
Statement = WhileStatement | IfStatement |
ExpressionStatement;
WhileStatement = “while” “(“ ConditionalExpression “)”
Block;
IfStatement = “if” “(“ ConditionalExpression “)” Block;
Block = “{“ (Statement)* “}”;
ConditionalExpression = <identifier>;
ExpressionStatement = <identifier>;

# code folding
FOLD:Block

# navigator support
NAVIGATOR:WhileStatement:”{$ConditionalExpression}”

# brace completion
COMPLETE “{:}”
COMPLETE “(:)”

# indentation support
INDENT “{:}”
INDENT “(:)”
INDENT “\\s*(((if|while)\\s*\\(|else\\s*|else\\s+if\\
s*\\(|for\\s*\\(.*\\))[^{;]*)”b
```

Blog by Jan Jancura, the lead NetBeans engineer for Schliemann

blogs.sun.com/hanz

project.

Once tokens are defined, one can already begin assigning features. For example, this single statement would fill the Navigator with the values provided by the “keyword” token:

```
NAVIGATOR:keyword
```

Readers who are familiar with the NetBeans Navigator API can only be amazed at this drastic simplification! However, normally you would like more robust support for a language and to provide a *grammar* in addition to tokens. The grammar that the Schliemann approach requires is also highly simplified. It is comparable to JavaCC or AntLR. Ideally, one would wish that the grammar provided by JavaCC and AntLR could be directly integrated into NetBeans IDE. Unfortunately, however, these grammars are not tailored to usage within an IDE. For this reason, a conversion process needs to take place, from AntLR or JavaCC (or from a similar approach) to the Schliemann NBS format.

Early experiments have shown that both a manual and an automatic solution for this process is feasible. However, this aspect of the Schliemann project is definitely the area where most work needs to be done. A unified, simple approach to integrating grammars provided by AntLR, JavaCC, and the like, is needed in order for the Schliemann project to reach its full potential.

In the NBS code shown before, you can see, in addition to the tokens, that the grammar forms the basis of both the Navigator implementation and the code folding implementation. In the case of code folding, the **Block** grammar definition determines each code fold, while the Navigator is populated by values conforming to the **WhileStatement** definition.

Finally, notice that the code also shows how brace completion and indentation is defined, all within the same single file, and that one can fine-tune further by specifying that white space should be skipped by the parser.

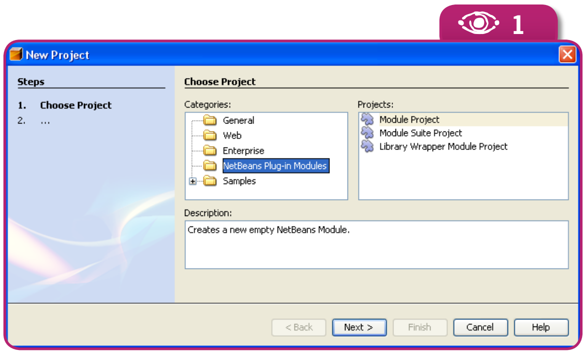
Hence, when the NBS file in **Listing 1** is associated with a MIME type, documents corresponding to the MIME type immediately have the following features:

- Syntax coloring
- Navigator
- Code folding
- Brace matching
- Indentation

In similar ways, a wide range of other language-support features can be created, including code completion, which is frequently very high up on the list of features that language programmers want to provide support for.

Getting started

Now that we have a general flavor of the Schliemann approach, let’s put it into



Creates a new empty NetBeans Module.

Figure 1
Creating a new module project.

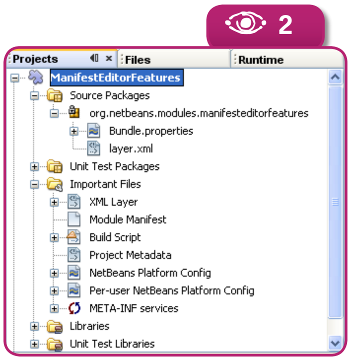


Figure 2
Result of the New Projects window: Plugin Source Structure

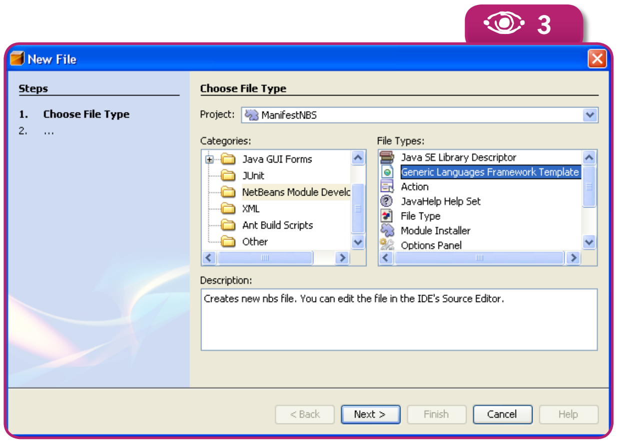
practice and create an NBS file for Java Manifests. Manifests, as you know, are constructed from key/value pairs. In the IDE, there is no language support for Manifests, not even syntax coloring. Let's provide that... and a lot more besides.

We begin as one always does when creating a plug-in for the IDE: by creating a new module project (see **Figure 1**). Next, in the New Project wizard, name the project "ManifestEditorFeatures" and specify "org.netbeans.modules.manifesteditorfeatures" as the Code Name Base. At the end of the wizard, after having clicked *Finish*, you'll see that the IDE has created a basic source structure, as it does for every NetBeans module (see **Figure 2**).

Next, we can use the Generic Languages Framework wizard to generate the NBS template discussed in the previous section. This template is found in the NetBeans Module Development section in the New File wizard (see **Figure 3**). Once you've completed the wizard, you have a single new file, in which we will do *all* our coding for this module (see **Figure 4**).

Now, let's begin! Unlike in the previous section, the syntax we are dealing with here has the notion of *state*. By state we mean that if we know in which token we find ourselves, we can always know where we are in relation to all the other tokens. So, for example, if we are in the "key" part of a key/value statement in a Manifest, we know that when we reach the colon we are entering the "value" part of the statement. As a result, we can define our tokens in the context of their *states*. Below you see how this is done. Not much of this should be foreign to you if you are familiar with regular expressions:

```
TOKEN:key:( [ ^ ”#”] [ ^ “:” “\n” “\r”]* ):<VALUE>
```



Project: ManifestNBS

Creates new nbs file. You can edit the file in the IDE's Source Editor.

Figure 3
Generic Languages Framework Template.

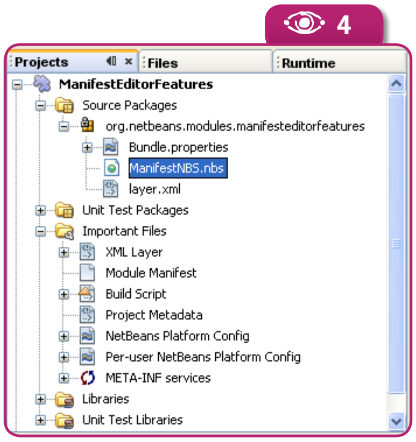


Figure 4
Result of the New File wizard: One additional file!

```
<VALUE> {
   TOKEN:whitespace:( ["\n" "\r"]+ ):<DEFAULT>
   TOKEN:operator:( ":" ):<IN_VALUE>
}
<IN_VALUE> {
   TOKEN:whitespace:( ["\n" "\r"]+ ):<DEFAULT>
   TOKEN:value:( [^ "\n" "\r"]* )
}
```

Notice that we start out by saying that we are *not* in a key if the first character is a hash (**#**). In that case we are, in fact, in a comment. It would also be good to provide a specific syntax color for comments, so let's define a token for comments:

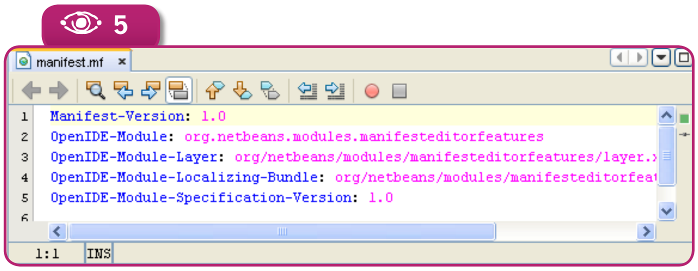
```
TOKEN:comment:( "#" [^ "\n" "\r"]* ["\n" "\r"]+ )
```

Right now, without going any further, we can already assign colors. Again we do so declaratively:

```
COLOR:key: {
   foreground_color: "blue";
}
COLOR:operator: {
   foreground_color: "black";
}
COLOR:value: {
   foreground_color: "magenta";
}
```

Apart from the foreground color, there are many other attributes that we can set per token, such as the style and background color. Without going much further, though, we can already install our module and then we'll have syntax coloring (see **Figure 5**)! It couldn't be much simpler. Before we do so, however, we need to create a MIME type resolver, which is a small XML file that specifies the file extension of the files we want to deal with.

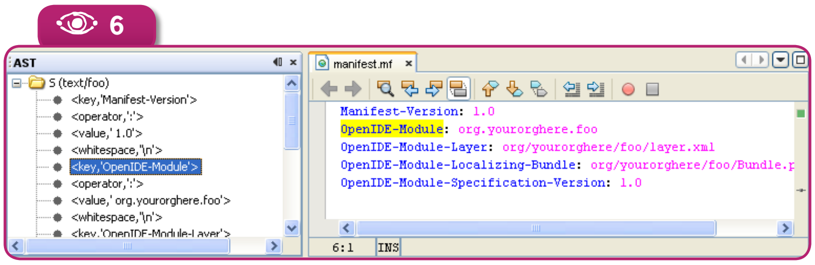
5



```
Manifest-Version: 1.0
OpenIDE-Module: org.netbeans.modules.manifesteditorfeatures
OpenIDE-Module-Layer: org/netbeans/modules/manifesteditorfeatures/layer.x
OpenIDE-Module-Localizing-Bundle: org/netbeans/modules/manifesteditorfea
OpenIDE-Module-Specification-Version: 1.0
```

Figure 5 A Manifest file with syntax coloring.

6

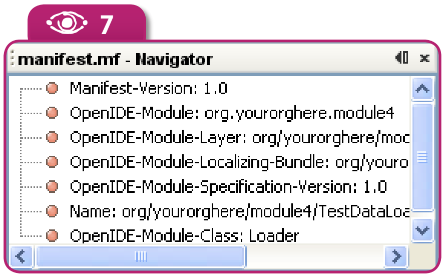


```
S (text/foo)
   <key,'Manifest-Version'>
   <operator,':'>
   <value,' 1.0'>
   <whitespace,'\n'>
   <key,'OpenIDE-Module'>
   <operator,':'>
   <value,' org.yourorghere.foo'>
   <whitespace,'\n'>
   <key,'OpenIDE-Module-Layer'>
```

```
Manifest-Version: 1.0
OpenIDE-Module: org.yourorghere.foo
OpenIDE-Module-Layer: org/yourorghere/foo/layer.xml
OpenIDE-Module-Localizing-Bundle: org/yourorghere/foo/Bundle.p
OpenIDE-Module-Specification-Version: 1.0
```

Figure 6 AST window.

7



```
Manifest-Version: 1.0
OpenIDE-Module: org.yourorghere.module4
OpenIDE-Module-Layer: org/yourorghere/moc
OpenIDE-Module-Localizing-Bundle: org/youro
OpenIDE-Module-Specification-Version: 1.0
Name: org/yourorghere/module4/TestDataLoa
OpenIDE-Module-Class: Loader
```

Figure 7 Navigator.

If you use the New File Type wizard, you can let the IDE generate such a MIME type resolver for you. You then need to register both the resolver and the NBS file in the XML layer file and declare a dependency on the Generic Languages Framework API. Eventually, the Generic Languages Framework template will do all of this for you, one imagines; but at the time of writing this is not the case.

After installing the module, we can develop it further. To help you, NetBeans 6.0 will provide a number of developer tools, such as the new AST window (see **Figure 6**), which lets you analyze a file, based on the tokens you have assigned to its MIME type. Ultimately, for Manifests, you could create a very detailed Navigator (see **Figure 7**), among other useful features for the end user.

Conclusions

Hopefully this broad introduction gives you a flavor of what NetBeans 6.0 will do for scripting languages. Quickly and without much fuss, language developers will be able to integrate their favorite scripting languages into the IDE, thus turning NetBeans more and more into their own, customized development environment. In short, just like Heinrich Schliemann, NetBeans IDE will be able to pick up new languages and expand its usefulness across more and more development communities.

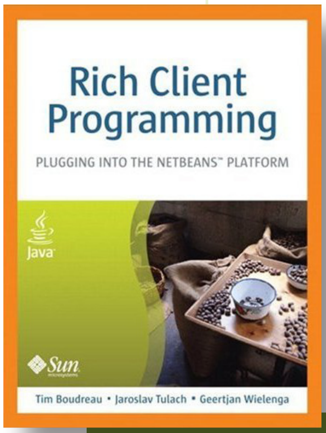
Geertjan Wielenga (*geertjan.wielenga@sun.com*) is a technical writer for NetBeans IDE and a co-author of the book "Rich Client Programming: Plugging into the NetBeans Platform". He is passionate about NetBeans and blogs about it daily at *blogs.sun.com/geertjan*.

A New Handbook for NetBeans Platform Development

"Welcome to the world of rich client development on the NetBeans Platform." So begins the new book on the NetBeans Platform, called "Rich Client Programming: Plugging into the NetBeans Platform". Written by three stalwarts of the NetBeans IDE, Tim Boudreau, Jaroslav Tulach, and Geertjan Wielenga, this new title from Prentice Hall introduces you to the central concepts of the NetBeans Platform.

The book begins by discussing the rationale for modular programming. "Loose coupling", whereby spaghetti code is avoided through a set of related but independent modules, is discussed with reference to a set of concrete examples. Gradually, the need for this approach to robust programming is introduced and, piece by piece, the authors highlight NetBeans' responses to this need. The tooling for modular programming since the release of NetBeans IDE 5.0 has proven its value to numerous programmers, and the reasons for this quickly become clear in the book. Typical stumbling blocks that new developers come across, such as "nodes" and "cookies" are explored in detail.

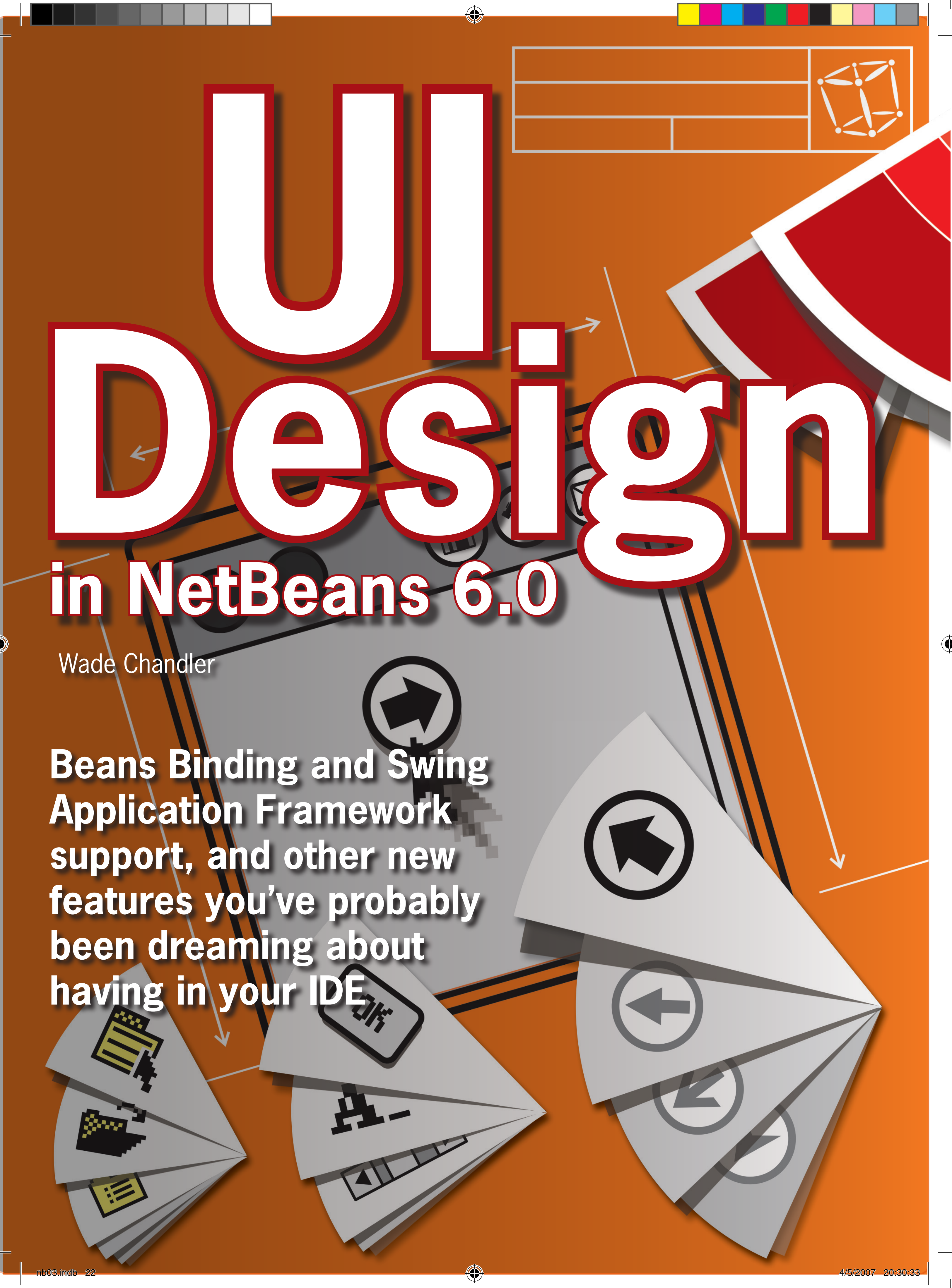
The second part of the book deals with a set of concrete NetBeans API scenarios. How, for example, can one provide code completion? Or hyperlinks in the Source Editor? And what about palettes with items that can be dragged and dropped? These and other scenarios are discussed in detail. Each chapter travels through a very specific example scenario, based on the example in the CD that accompanies the book. At the end of this part, the reader should understand some of the basic APIs that are typically implemented by developers making use of the NetBeans Platform.


Rich Client Programming
PLUGGING INTO THE NETBEANS™ PLATFORM
Java
Sun
Tim Boudreau • Jaroslav Tulach • Geertjan Wielenga

Rich Client Programming: Plugging into the NetBeans™ Platform
Tim Boudreau,
Jaroslav Tulach,
Geertjan Wielenga
(Prentice Hall)
ISBN-13: 978-0-13-235480-6
640 pages

The book closes with two chapters contributed by two developers with years of experience with the NetBeans Platform. First, Jens Trapp, from Germany, discusses how he integrated the HTML Tidy project into NetBeans IDE. In doing so, he brings together many of the principles and APIs discussed in the preceding chapters. Next, US-based Rich Unger describes a complete application built on top of the NetBeans Platform, for editing WAV files. Typical concerns involved in creating Platform-based applications are discussed in this chapter. Together Jens and Rich provide the two "use cases" of the NetBeans Platform – allowing you to extend NetBeans IDE with new features, and creating completely separate applications, which in turn could constitute the platform of still other applications.

This is the first book since Tim Boudreau's and Jesse Glick's "NetBeans: The Definitive Guide" – the popular title from some years ago which had a large section on the NetBeans Platform – to cover the length and breadth of the NetBeans Platform. Judging from the level of interest shown in the Safari Rough Cuts version of the book, which provides an early draft in PDF format, "Rich Client Programming: Plugging into the NetBeans Platform" promises to take the Swing development world by storm. If you want to leverage the full potential of the NetBeans Platform, this is definitely not a book that you want to miss!



UI Design

in NetBeans 6.0

Wade Chandler

Beans Binding and Swing Application Framework support, and other new features you've probably been dreaming about having in your IDE

One of the most talked about and innovative features since NetBeans 5.0 is Project Matisse or the Form Editor. Many would agree that Matisse is the best user interface designer across different IDE categories and technologies. A good number of developers will start or have started using NetBeans because of it.

With software, there is never-ending room for improvement and growth. In NetBeans 6.0, familiar and new developers will find many new Matisse features to help improve Swing application development, as you'll see in this article.

Beans Binding and the Swing Application Framework

Building on the recent JCP specs Beans Binding (JSR 295) and Swing Application Framework (JSR 296), desktop developers have a few new cards up their sleeve; they'll also benefit from more efficient development. Plain Swing application developers gain more from the Swing Application Framework enhancements in NetBeans, but Platform developers should not feel left out. The latter already have a Swing-based application framework in the NetBeans Platform, with many more features than JSR 296 provides. All gain much with Beans Binding support, however.

Beans Binding support

Beans Binding allows you to escape the common monotony of writing code to copy data from user interface components to data classes or JavaBeans and vice versa. With the new Beans Binding enhancements, you can right click on a UI element and access the *Bind* context menu item. For instance, selecting this item for a **javax.swing.JTextField** will show the preferred bound property of *text* (see **Figure 1**). Other properties can be accessed through the

form.netbeans.org

Matisse/Form Editor project home page

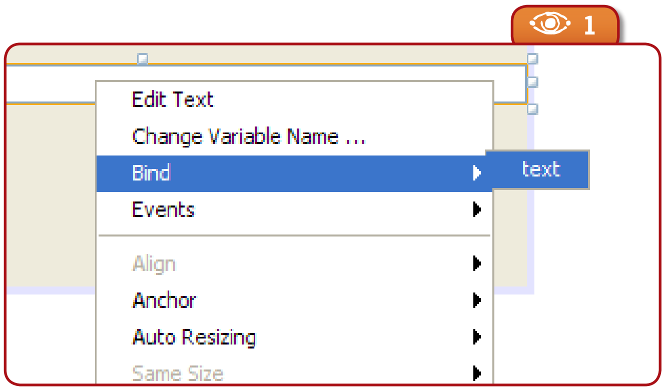
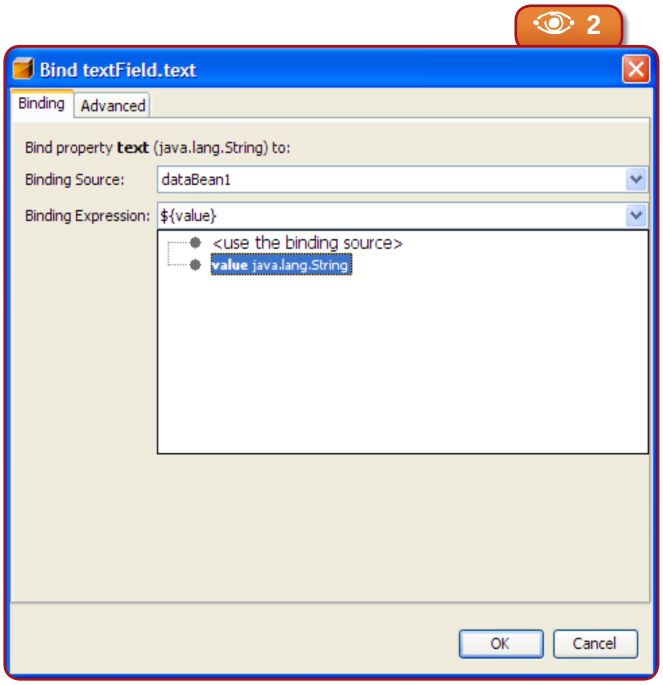


Figure 1
The Beans Binding menu item.



Bind property **text** (java.lang.String) to:

Binding Source: dataBean1

Binding Expression: ${value}

<use the binding source>
value java.lang.String

Figure 2
Setting Beans Binding options for a JTextField.

property inspectors Binding tab (**Figure 2**).

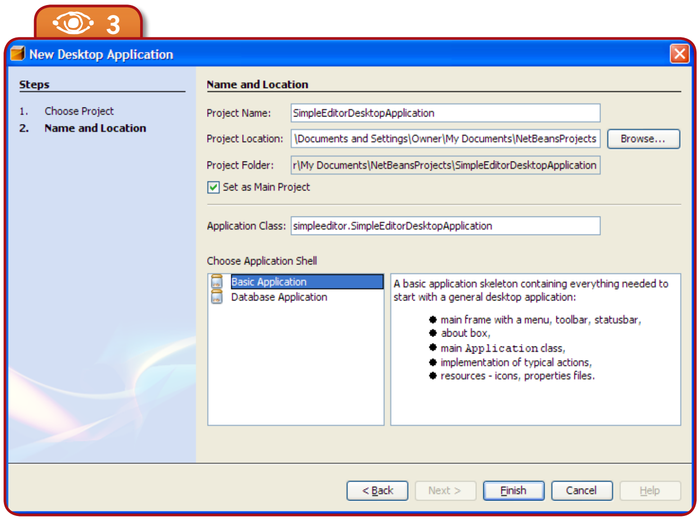
Once you've chosen the property to bind, a dialog is displayed, where you can select the target JavaBean. You can then enter an expression using the Beans Binding Expression Language. The syntax is much like the JSP EL. Aside from entering the expression by hand, a nifty selector, accessed as a drop-down list or combo box, allows developers to quickly build the expression by drilling down through properties and sub-properties.

When the application is run, the selected UI component updates its bound component when focus is lost or the user presses Enter. The main point is that the developer no longer has to manage this and other operations with a load of event handling code.

Swing Application Framework support

The Swing Application Framework provides mechanisms to quickly build complete desktop applications. NetBeans 6 takes it further with integrated support for the framework in the IDE, while providing a set of standard icons such as Copy, Paste, and Cut. This is better than piece milling an application from Swing components, and coming up every time with a separate solution for starting an application or shutting it down, basic actions or events, custom icons, a resource manager, session storage, etc.

To create a Swing Application Framework project in NetBeans 6, select *File|New Project* and, under the General category, choose the new Java Desktop Application project template. At the time of writing, there are two application templates available: Basic and Database (see **Figure 3**). Others will be included in future releases. For example, thought is being given to templates for creating web client applications based on the Swing Application Framework.



1. Choose Project
2. **Name and Location**

Name and Location

Project Name: SimpleEditorDesktopApplication

Project Location: \Documents and Settings\Owner\My Documents\NetBeansProjects

Project Folder: r\My Documents\NetBeansProjects\SimpleEditorDesktopApplication

☑ Set as Main Project

Application Class: simpleeditor.SimpleEditorDesktopApplication

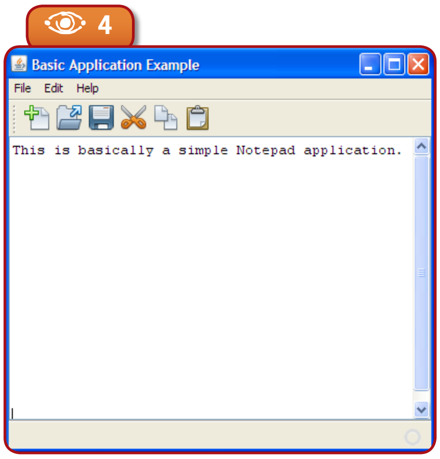
Choose Application Shell

- Basic Application
- Database Application

A basic application skeleton containing everything needed to start with a general desktop application:

- main frame with a menu, toolbar, statusbar,
- about box,
- main `Application` class,
- implementation of typical actions,
- resources - icons, properties files.

Figure 3 Choosing a desktop application template.



This is basically a simple Notepad application.

Figure 4 Example application generated with the Basic template.

The Basic template generates a regular Swing application with simple features such as Cut, Copy, Paste, Save, and New. Simple applications like Notepad or KWrite can be easily created with this template (see **Figure 4**).

The Database template allows users to create CRUD database applications. Along with Apache Derby/JavaDB or HSQLDB, this can be like Microsoft Access on steroids.

To me, the Swing Application Framework support works best for developing simpler applications. This may get better in the future with JSR 277 (Java Module System). However, a module system alone does not provide a framework with all the components and utilities offered by the NetBeans Platform. For more complex applications, I advise you to build on the NetBeans Platform. A path from Swing Application Framework-based applications to Platform-based apps is being pondered, but nothing is concrete in this regard at the time of writing.

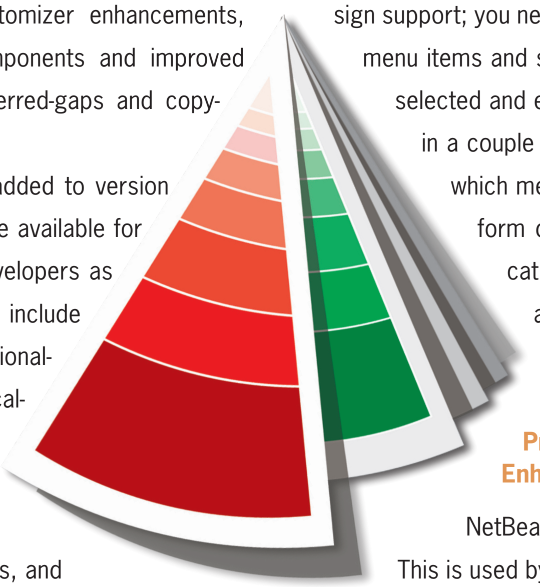
The Swing Application Framework and the NetBeans Platform are both Swing frameworks. The Swing Application Framework provides application lifecycle management, session management (e.g.

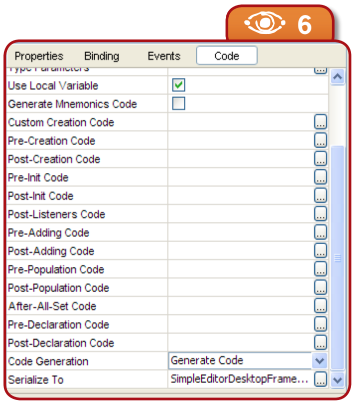
windows are stored in the same locations when the application restarts), a resource manager, actions, storage, an application context, and synchronous and asynchronous tasks. The NetBeans Platform provides all these plus numerous other features, including a powerful module/plug-in system which allows modules to install their own UI menus, actions, and services, among other application items.

More new features

Additional new features in NetBeans 6 include a new visual menu designer, protected code customizer enhancements, centering of components and improved Free Design preferred-gaps and copy-paste support.

Other features added to version 6 have been made available for NetBeans 5.5 developers as an update. These include automatic internationalization, visual localization, a preview with look and feel support, relative font definitions, and dragging of components from the projects explorer to the UI – as well as a context-sensitive help bar, and a pre/post declaration code editor. In the upcoming IDE release, all these features will be fully integrated, and come out of the box.





Use Local Variable	☑
Generate Mnemonics Code	□
Custom Creation Code	...
Pre-Creation Code	...
Post-Creation Code	...
Pre-Init Code	...
Post-Init Code	...
Post-Listeners Code	...
Pre-Adding Code	...
Post-Adding Code	...
Pre-Population Code	...
Post-Population Code	...
After-All-Set Code	...
Pre-Declaration Code	...
Post-Declaration Code	...
Code Generation	Generate Code
Serialize To	SimpleEditorDesktopFrame...

Figure 6 Protected Code Customizer enhancements.

Visual Menu Designer

Previous versions of Matisse had limited application menu design support; you needed to use the Inspector window to create menu items and sub-menus. Now menu components can be selected and edited visually in the UI designer. This helps in a couple Java Desktop Application ways: it's clearer which menu and menu item is being edited, and the form doesn't have to be previewed or the application run to see what the menu will look like at runtime. See the Visual Menu Designer in action in **Figure 5**.

Protected Code Customizer Enhancements

NetBeans uses the concept of protected code. This is used by Matisse so that the generated code is not changed and possibly broken by the developer, allowing the visual designer to continue working. Sometimes, though, this is a little too strict, and restrictions on protected code are a common complaint in the NetBeans mailing lists.

In NetBeans 6 this will be much better, and you'll be able to change the protected code sections in many ways. These help you get around issues of timing actions and method calls, with initialization and property/bean configuration, for example.

Here are some options for modifying protected code, all accessible through the Properties window Code tab

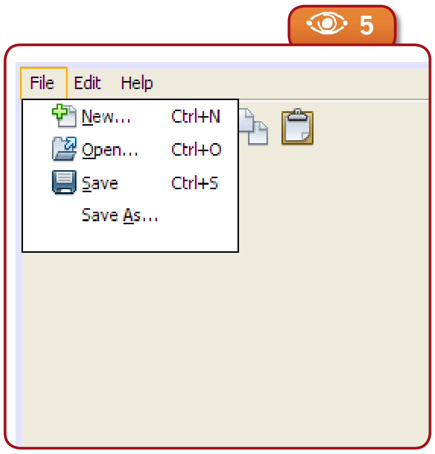
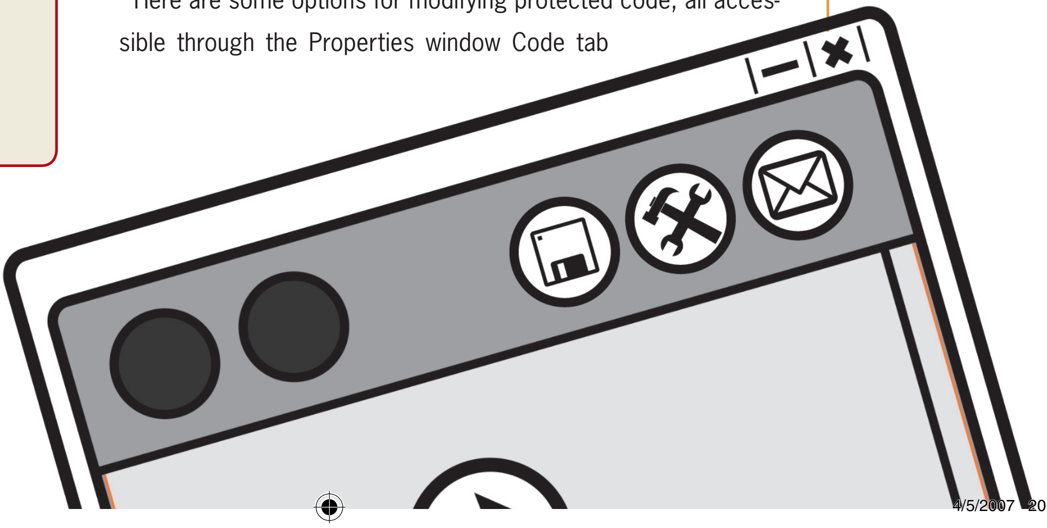
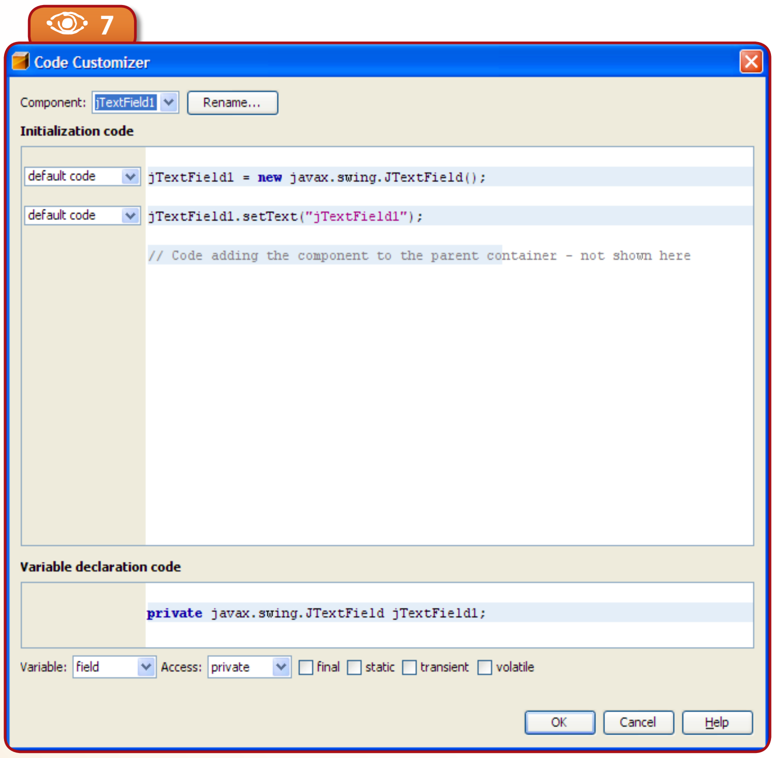


Figure 5 New Visual Menu Designer.

wiki.netbeans.org/wiki/view/NetBeansDreamTeam

NetBeans Dream Team Wiki/home page





Code Customizer

Component: jTextField1 Rename...

Initialization code

```
default code   jTextField1 = new javax.swing.JTextField();
default code   jTextField1.setText("jTextField1");
               // Code adding the component to the parent container - not shown here
```

Variable declaration code

```
private javax.swing.JTextField jTextField1;
```

Variable: field Access: private □ final □ static □ transient □ volatile

Figure 7 Code Customizer Dialog

Swing Application Framework and Beans Binding demo by Roman Strobl

blogs.sun.com/roumen/entry/swing_application_framework_swing_databinding

(also see **Figure 6**):

- *Post-Listener-Code* – Included after all properties of all beans are set and all listeners are added.
- *Pre-Adding Code and Post-Adding Code* – Included before and after the component is added to the parent container such as a **JPanel** or **JFrame**.
- *After-All-Set Code* – Included after the component is completely set-up.

The new Pre/Post Declaration Code Editor is more of a must-have than a development boost. As of Java 5, developers can now annotate different things in Java source code, and for libraries or technologies requiring annotations, Matisse must allow the developer to somehow set these annotations. Annotations can also be added through the Properties window's Code tab, in the Pre-Declaration Code field; and there's a Post-Declaration Code field available.

NetBeans 6 comes with a new code customizer, which lets you inject source code more easily into the protected sections. It's accessible by right clicking on the UI form in the designer and

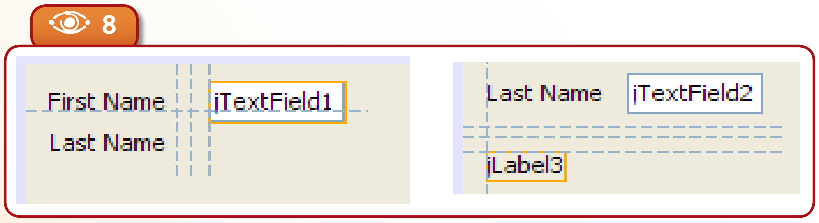
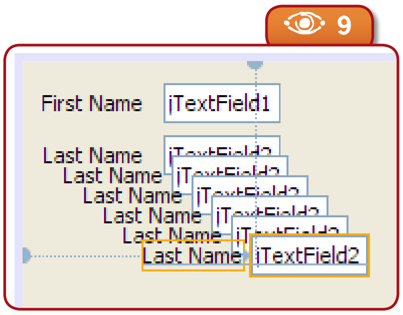


Figure 8 Free Design preferred gap support – three offsets instead of one



9

selecting *Customize Code*. A dialog with a Java editor pops up. See **Figure 7** for an example.

Centering of components

Aligning components along a center axis just got easier. Matisse now allows a group of selected components to be centered down the axis of the first selected component. Horizontal and vertical centering are available. Currently, more than one component must be selected; then all are centered on the widest selection. Another option will allow centering components horizontally and vertically in their parent container.

More gaps

Before NetBeans 6, the Free Design layout manager supported a single preferred gap for component placement. In 6.0, three preferred gaps are supported. A preferred gap is the preferred spacing between components, and is available on all sides of a component for quick and elegant placement, as shown in **Figure 8**. For developers who may like more control over spacing between components, having three choices comes in handy.

Better copy/paste support

While using the Free Design layout, copying and pasting produces a horrible result in versions before 6.0. It places all copied components, no matter the number selected, at location [0,0] on the UI form. This problem has now been solved. Instead of messing up the layout, copies are now pasted to a manageable offset of the original components. The components also keep their layout after copying (see **Figure 9**).

Automatic Internationalization and Visual Localization

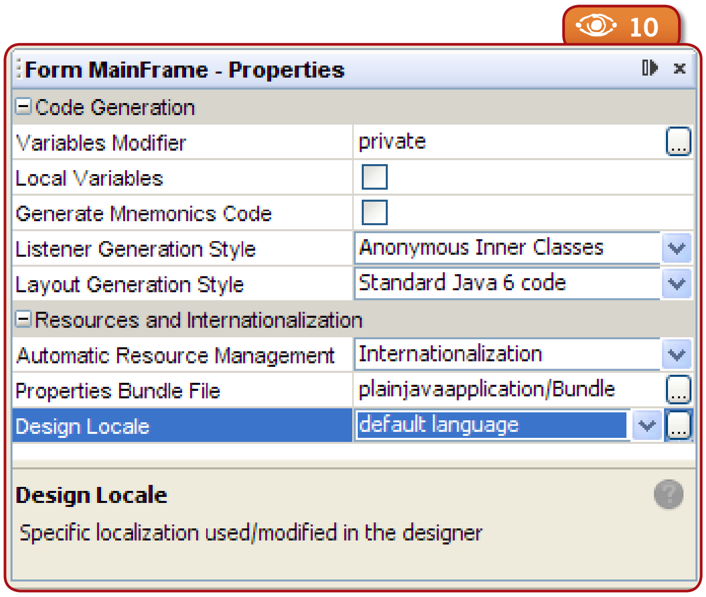
Previous releases of NetBeans required a few extra steps to internationalize and localize a UI. Now the process is streamlined. In previous versions, you had to setup each individual UI element to pull values from the correct resource bundle. Then to actually localize the bundle you needed to create separate bundle files or localized entries manually, enter the text for the correct locale, and format the file accordingly (using a different encoding, for example).

In NetBeans 6.0, you can now ask Matisse to automatically internationalize the application during UI design. This means that for each resource that would normally be internationalized by hand Matisse automatically adds the value supplied in the UI designer to a resource bundle. This is done for all UI elements.

The visual localization feature works in harmony with automatic internationalization. You can right click the top form node in the tree of the Inspector window, locate the Design Locale combo box in the Properties window, and then select a locale or add a new one.

Once the Design Locale is selected, you can simply edit the text

10



Form MainFrame - Properties

⊟ Code Generation	
Variables Modifier	private
Local Variables	□
Generate Mnemonics Code	□
Listener Generation Style	Anonymous Inner Classes
Layout Generation Style	Standard Java 6 code
⊟ Resources and Internationalization	
Automatic Resource Management	Internationalization
Properties Bundle File	plainjavaapplication/Bundle
Design Locale	default language

Design Locale

Specific localization used/modified in the designer

Figure 9
Improved copy/paste support in Free Design (components were pasted multiple times).

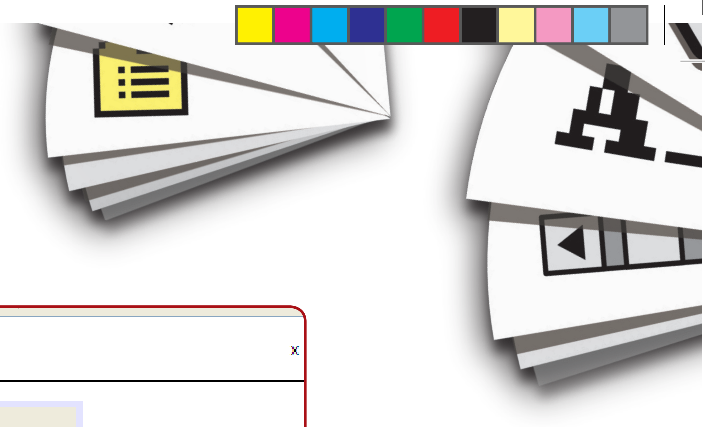
wadechandler.no-ip.com/~wchan

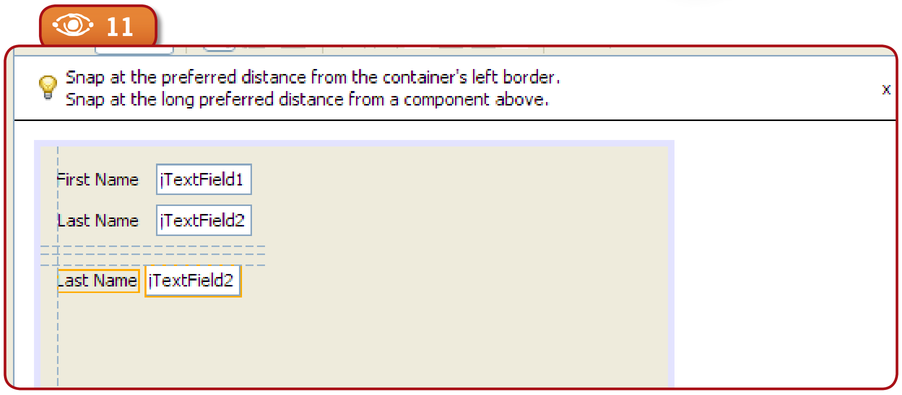
Wade Chandler's home page

appframework.dev.java.net

A JSR 296 Swing Application Framework implementation

Figure 10
Access the Automatic Resource Management combo box to set the resource management style to Internationalization, and Design Locale to set the locale used while editing the UI in Matisse. Change the locale and re-edit the UI to localize the user interface. Nice and easy!





Snap at the preferred distance from the container's left border.
Snap at the long preferred distance from a component above.

First Name jTextField1
Last Name jTextField2
Last Name jTextField2

Figure 11 Context-sensitive help bar

NetBeans 5.5 Matisse update feature page
form.netbeans.org/updates/update55.html

in the UI as if normally editing a form. The localization for the Design Locale takes place automatically with the values entered into the UI. The locale can be changed and the form re-edited to set the values for the newly selected locale. It is as simple as it sounds to create a fully internationalized UI with NetBeans 6 (see **Figure 10**).

Preview with look and feel support

Before NetBeans 6.0, there was no mechanism to allow the project's look and feel to be set in the designer for a regular Swing application. It was left to the developer to add the appropriate code to set the application look and feel.

Setting the look and feel of a regular Swing application with source code is quite simple. However, without the Form Editor allowing the design look and feel to be set, the developer has no way to see how the application will look at runtime with different look and feels. This will no longer be a problem.

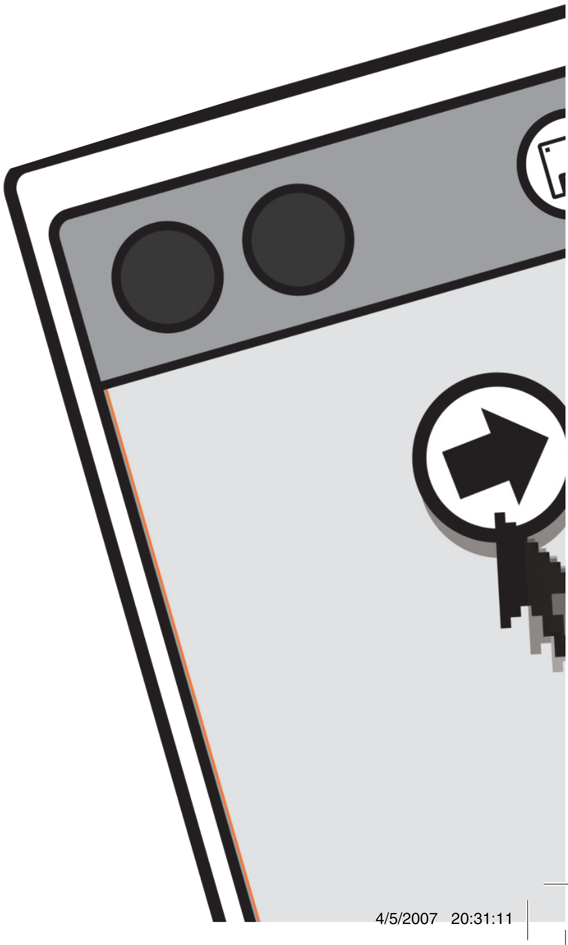
In NetBeans 6.0, forms can be previewed with any available look and feel. In the Inspector window, the developer right clicks on the form containers node, or on the node on the same level as the Other Components node and just below it (its sibling), which has the name of the forms extended class such as **JFrame**, **JDialog**, or **JPanel**. Then the Preview Design sub-menu is rolled over and the look and feel is selected. The form is then previewed with the chosen look and feel.

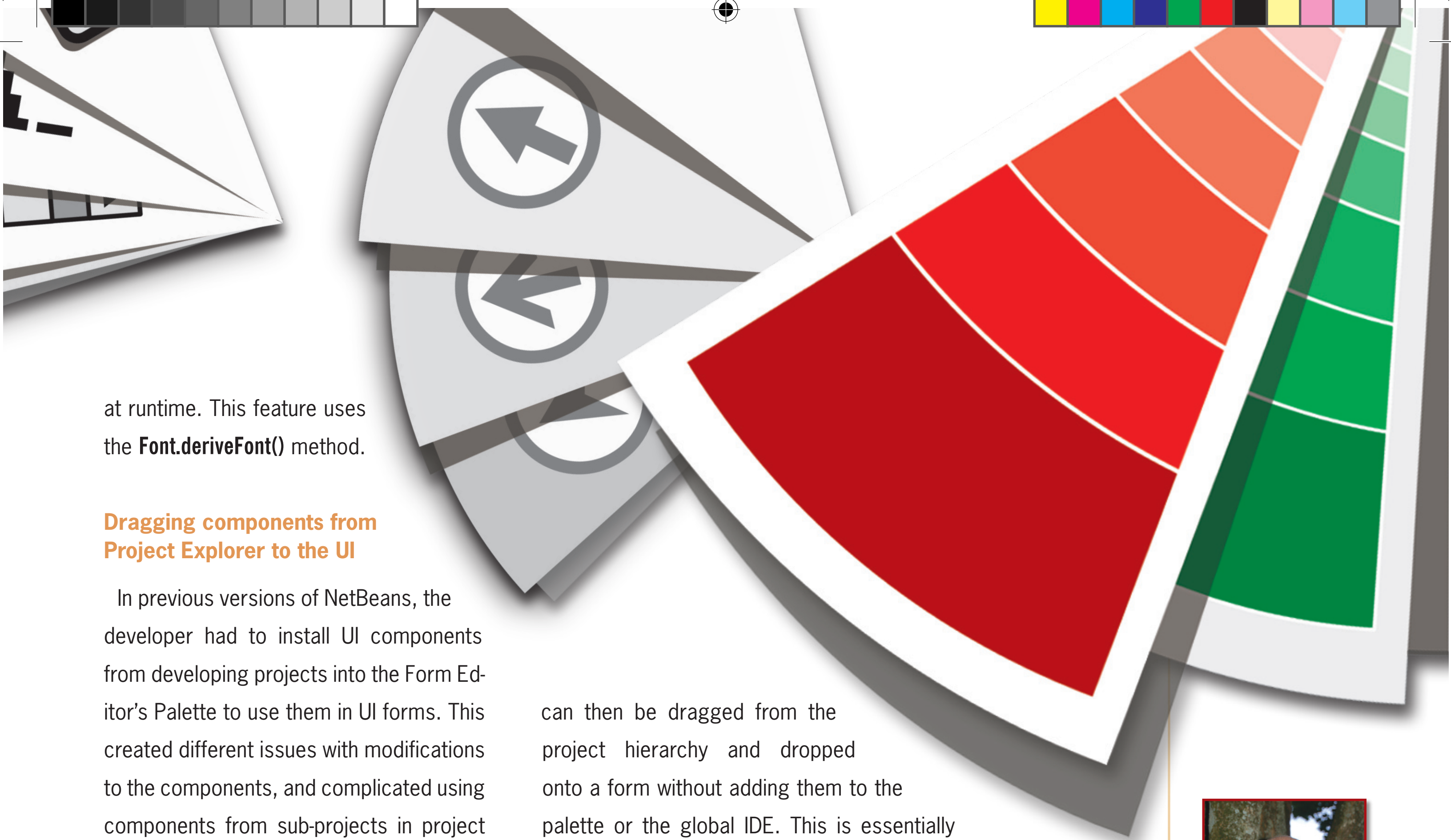
A right-click on the form in the UI designer also makes the Preview Design sub-menu accessible.

Relative font definitions

Have you ever wanted to define your UI fonts just a little larger or smaller than the overall system or application font? This would come in handy in that fonts would not have to be set for each component. Instead, fonts for specific components could be set to a different relative size, or have a different style than the main application or system font.

In the latest version of Matisse this is quick and simple. You select a UI component on a form, and in the Properties window click the button to the right of the **font** property. The font editor appears. There, you can select the checkbox "Derive the font from the default font" and choose the font style. The font can be sized relative to the default font; the size can also be made absolute. Regardless of the size, the font is not hard-coded and will be determined



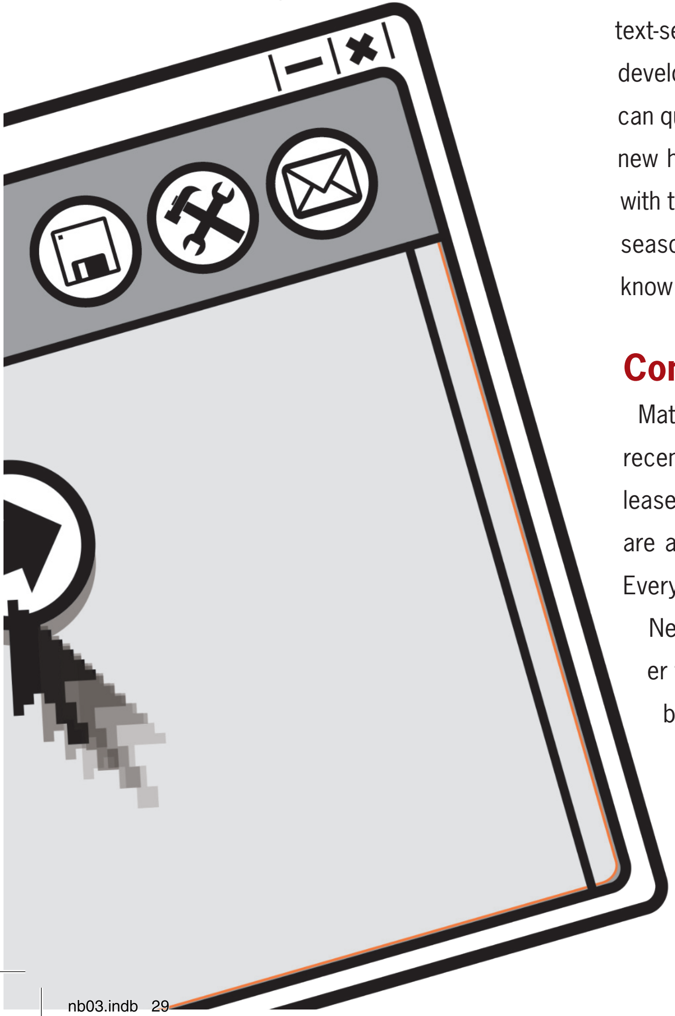


at runtime. This feature uses the **Font.deriveFont()** method.

Dragging components from Project Explorer to the UI

In previous versions of NetBeans, the developer had to install UI components from developing projects into the Form Editor's Palette to use them in UI forms. This created different issues with modifications to the components, and complicated using components from sub-projects in project UIs. Things have gotten better, however.

Now you can work on a library as a sub-project or dependency. You create a new form in the parent project, and the sub-project UI controls/classes can then be dragged from the project hierarchy and dropped onto a form without adding them to the palette or the global IDE. This is essentially project-level palette items: a nicer and cleaner solution.



Context-sensitive Help Bar

Who says you can't teach an old dog new tricks? The new context-sensitive help bar (see **Figure 11**) can help new and familiar developers alike. New users will probably find it more useful, as it can quickly bring them up to speed with the Matisse features. The new help bar provides contextual hints about what can be done with the selected component or current feature being used. Even seasoned users may find new shortcuts or features they did not know existed.

Conclusions

Matisse is part of the reason for the NetBeans community's recent growth spurt, and it gets better and better with each release. The 6.0 release is no different, and the latest features are a great boost to Swing application development efficiency. Everyone should try it.

NetBeans 6.0 Matisse feature development is ongoing, and other features are currently in the works. Most notably, the possibility to fix refactorings for generated UI code should be ready in the final version, along with many other new features and improvements.

Wade Chandler (*hwadechandler-nb@yahoo.com*) is a software engineer working for Decision Dynamics, Inc. and an independent software developer. He began his career in 1997 and has been involved with the NetBeans community since it was known as Forte for Java, and with Java since JDK 1.0. Wade contributes to the NetBeans project, and is a member of the NetBeans Dream Team.

Write Once Deploy Anywhere

Anatole Wilson

Writing Applications for mobile devices and reducing device fragmentation with NetBeans Mobility Pack

Take a look around you in just about any location – your office, walking down the street, the movies, even your own household – and it won't take a marketing genius to tell you that if your application is useful to people on the go but hasn't been ported to phones or PDAs, you're missing a pretty large market, potentially *millions* of users. Of course, that same look around will tell you that your game or widget had better be able to work on mobile devices of all names and sizes, or you'll cut out a large portion of your potential customer base.

Sure, it sounds like a lot of work. But in this article, we'll show you how to use the NetBeans Mobility Pack 5.5 for CLDC/MIDP to create a simple application suitable for cell phones and then deploy it to two very different devices. Best of all, you'll do it using only a single set of code instead of creating and managing a separate code base for each device.

In addition, we'll take a quick look at the Mobility Pack 5.5.1 for CDC and show you how you can use it to develop applications for larger devices, like set-top boxes, PDAs, and the new generations of "smart" phones.

A brief overview of the Mobility Pack

With NetBeans 5.5, there are now two separate versions of Mobility Pack – one for each of the standard *configurations* that specify the minimum requirements for memory, Java language features, JVM support, and runtime libraries.

In Mobility Pack 6.0, the CLDC/MIDP and CDC Packs will be merged.

The Mobility Pack for Connected, Limited Device Configuration / Mobile Information Device Profile (CLDC/MIDP) serves developers who create applications targeted at smaller mobile devices with limited resources, such as mobile phones, PDAs, and two-way pagers.

The Mobile Information Device Profile is one of two profiles that works "on top" of CLDC (the other profile being the Information Module Profile). MIDP provides graphical interfaces for interactive applications and is the standard profile for mobile telephone development under the JSR 185, Java Technology for the Wireless Industry (JWTI) specification, and more recently also the Mobile Service Architecture (MSA) spec, JSR 248.

The Mobility Pack for CDC provides a development environment for developers who are creating applications for larger devices, such as set-top boxes, embedded devices, high-end or smart phones, and PDAs. The CDC Pack supports development for all the current CDC Profiles, including the Foundation and Personal profiles, as well as the AGUI toolkit. It also supports development for the Windows CE platform.

netbeans.org/kb/trails/mobility.html

The Mobile Applications Learning Trail guides you through a structured approach to learning about the Mobility Pack.

How the two Java ME configurations fit into the Java "family" is shown in **Figure 1**.

In the next section, we'll create a MIDP application, or *MIDlet*, using the Mobility Pack for CLDC/MIDP. Then we'll look at how you can create an application using the Mobility Pack for CDC. It's worth noting here that with version 6.0, the two Packs will be integrated into a single Mobility module.

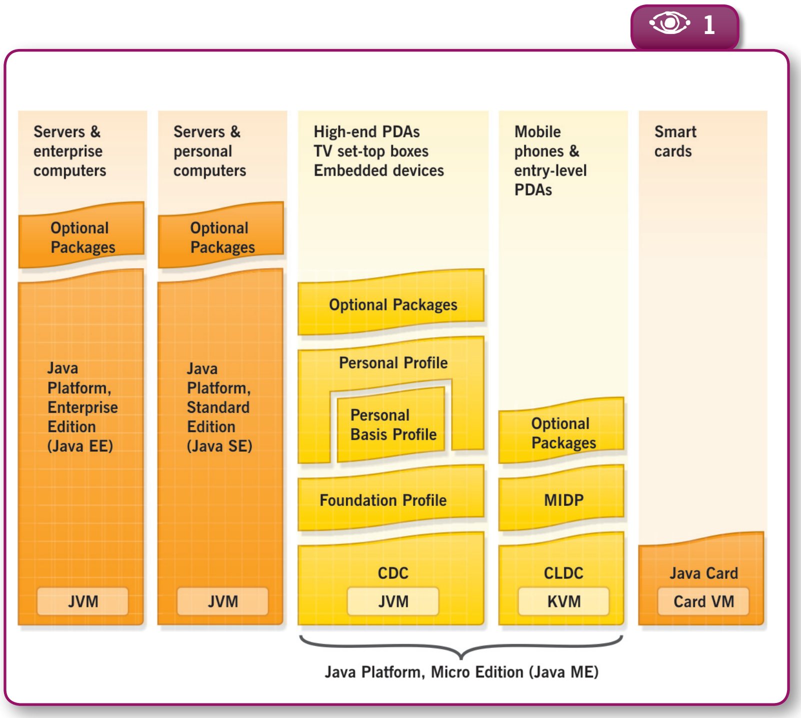

1
Servers & enterprise computers
Optional Packages
Java Platform, Enterprise Edition (Java EE)
JVM
Servers & personal computers
Optional Packages
Java Platform, Standard Edition (Java SE)
JVM
High-end PDAs TV set-top boxes Embedded devices
Optional Packages
Personal Profile
Personal Basis Profile
Foundation Profile
CDC
JVM
Mobile phones & entry-level PDAs
Optional Packages
MIDP
CLDC
KVM
Smart cards
Java Card
Card VM
Java Platform, Micro Edition (Java ME)

Figure 1 Java ME and the Java technology landscape.

Creating a MIDlet

We're going to start with a very simple "Hello World" MIDlet. Although this is about the simplest MIDlet you can create, it will let us demonstrate the Mobility Pack without too much focus on writing code.

To begin working with the Mobility Pack, you create a new project just as you would with any Java application, only selecting a Mobile project type.

In the NetBeans IDE, as you know, all Java development has to take place within a project. The IDE builds its project infrastructure directly on top of Apache Ant, and stores all the information about your project in an Ant script, a properties file, and a few XML configuration files. This means that you can build and run your projects outside of the IDE in exactly the same way as you would inside it. As you can see in the source tree shown in **Figure 2**, however, a MIDP project differs from a NetBeans/Java SE project in several ways.

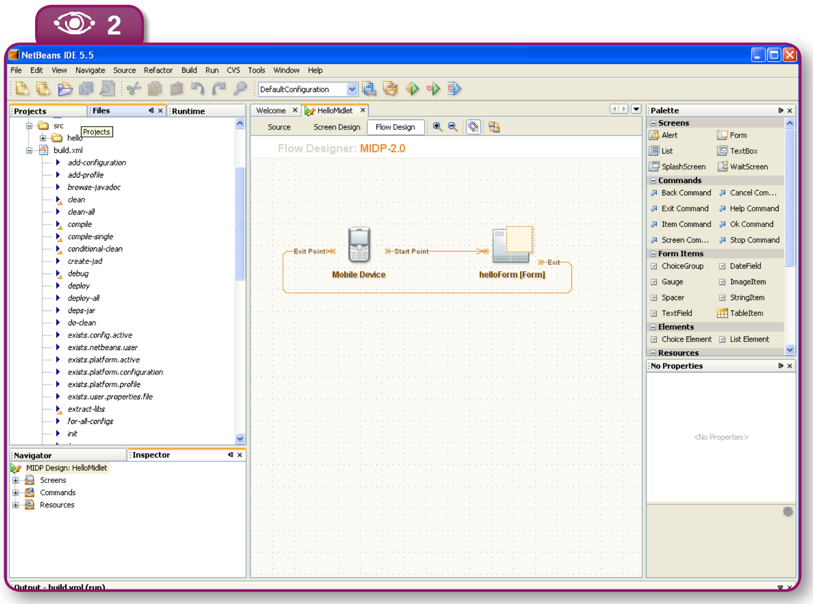


Figure 2 The Hello MIDlet Application .

When a MIDP project is created, the IDE creates folders to hold the files after they are compiled and preverified. (Preverification is a Java ME-specific building step that augments *.class* files with extra information that makes bytecode verification and classloading much simpler and faster on the resource-restricted JVMs of ME devices.) The preverified files are packaged into the distribution JAR of your project. If you use project

3

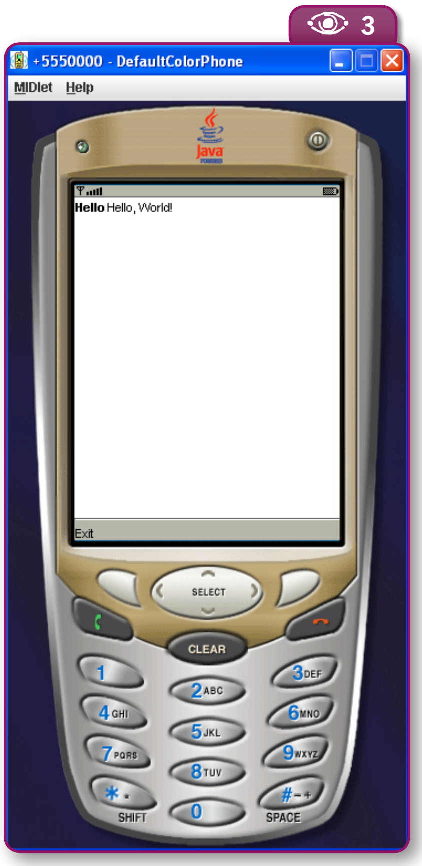


Figure 3
Hello MIDlet in action.

configurations for different device types (we'll get to that soon) or use obfuscation, the IDE creates folders to hold the different versions of the source files.

Now let's create the Hello MIDlet project. Choose *File>New Project* and in the first step of the wizard select the *Mobile* category and *Mobile Application* for the project type. Then click *Next*. Change the project name to "TestMIDlet." Click *Next* again to see the default Emulator Platform, which is the J2ME Wireless Toolkit. The default device is the DefaultColorPhone.

Click *Next* to see some other project configurations you can choose to set. The IDE checks all of the platform emulators you have installed and creates a configuration template for each device emulator available. Check the "MediaControlSkin_template" box. Click *Finish* and you've just created your first working MIDlet, containing a simple form named **helloForm**, that displays the familiar "Hello, world!" message. Now Choose *Run>Run Main Project* to see the MIDlet in action. You'll see a device emulator like the one shown in **Figure 3**.

Designing flow and UI in the Visual Mobile Designer

The screenshot in **Figure 2** also shows you the flow of our MIDlet in the Visual Mobile Designer (VMD). You can click the *Source* button and edit the code manually, but the VMD is a very powerful tool for designing both the page flow and individual screens.

Let's look at the flow first. If you start with the mobile device on the screen, it's pretty easy to follow the arrows and understand the flow of the MIDlet. When a command is entered (the Start Point), the **helloForm** is opened. When the exit command is chosen from this form, you close your application (Exit Point) and return to the mobile device's application launcher.

Now, you might want to add a splash screen to display your logo or development credits before the application starts running. Let's add a simple one to this application. (To provide a graphic, we've added *mobileduke.png* to our *<project_source>/src* directory. The

4

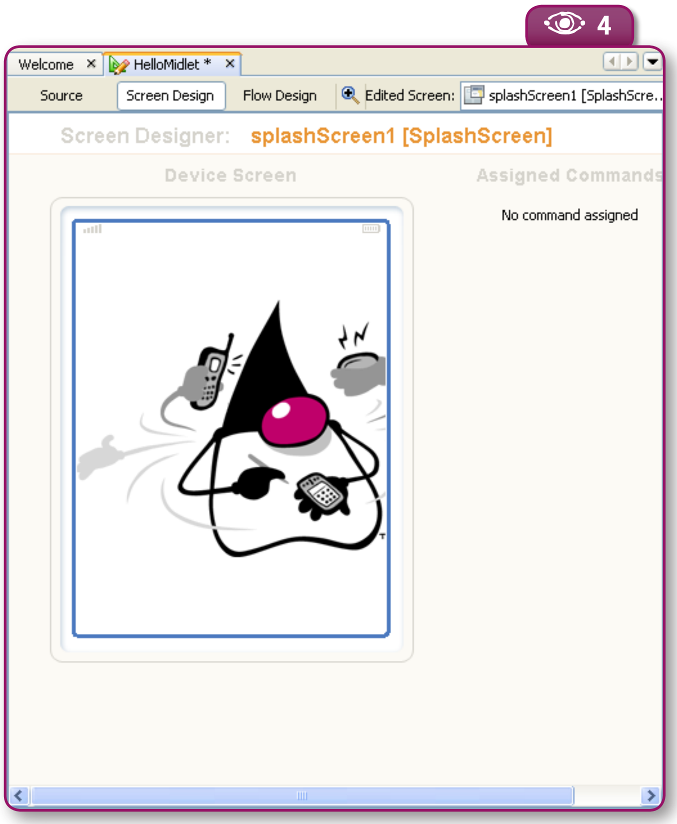


Figure 4
A splash screen in the Screen Designer.

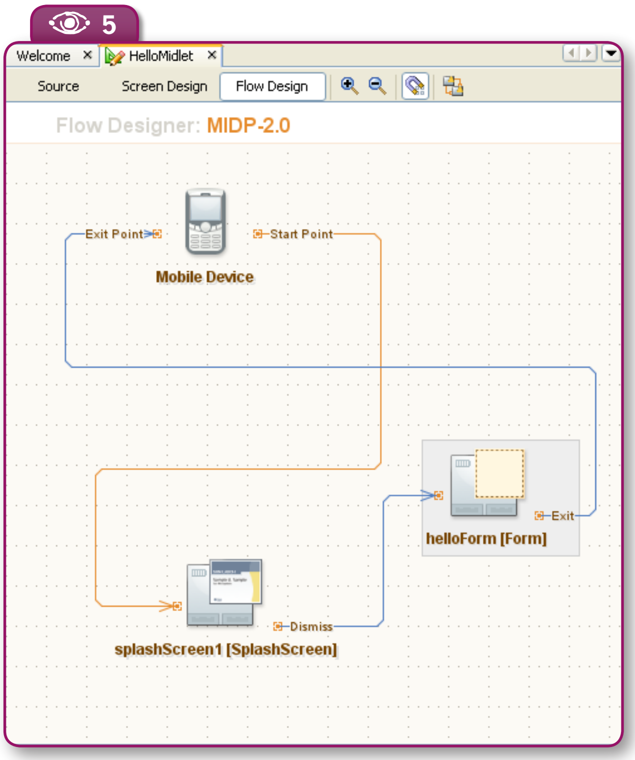


Figure 5 Application flow with a splash screen.

IDE automatically makes it available to us from the *Resources* folder in the Navigator window, on the lower left side of the IDE.) Drag a SplashScreen component from the Tools Palette on the right of the window on to the Flow Designer. Next, Drag the **image1[Image]** from the Navigator window and drag it on top of the splash screen.

Presto! Our splash screen has been created. You can look at it by double-clicking on the SplashScreen component, which takes you into the Screen Designer. The splash screen should look like **Figure 4**.

Now let's make the splash screen part of the application flow. Click the Flow Design button to go back to the Flow Designer. Grab the tip of the Start Point Arrow and drag it over to the **splashScreen1** component. Click on the orange square next to Dismiss and drag the arrow over the **helloForm** component. The flow should now look something like **Figure 5.**

Let's just make one more quick change to the program to illustrate the Screen Designer. Double click on the form component. In the Screen Designer, click on the "Hello, World!" text and change it to something else, say "Hello, Universe!"

Now go ahead and run the MIDlet again. Click the button under "Launch," and the emulator displays the splash screen as shown in **Figure 6**. Then you see the new message.

That was just a simple example of how you can quickly design the screens and flow of your application in the VMD. The SplashScreen component is one of three custom components, along with WaitScreen and TableItem, created by the NetBeans Mobility Pack team and added to the palette to make visual programming easier. You can also create your own custom components and add them to the palette.

Another important VMD feature is its support for Scalable Vector Graphics (SVG). SVG is an XML-based standard defined by the W3C and supported by MIDP 2.0 through JSR 226. The compact size and consistent appearance across different platforms and screen resolutions makes it an attractive graphics format for mobile developers. SVG also enables scripting and animation that allows users to interact with

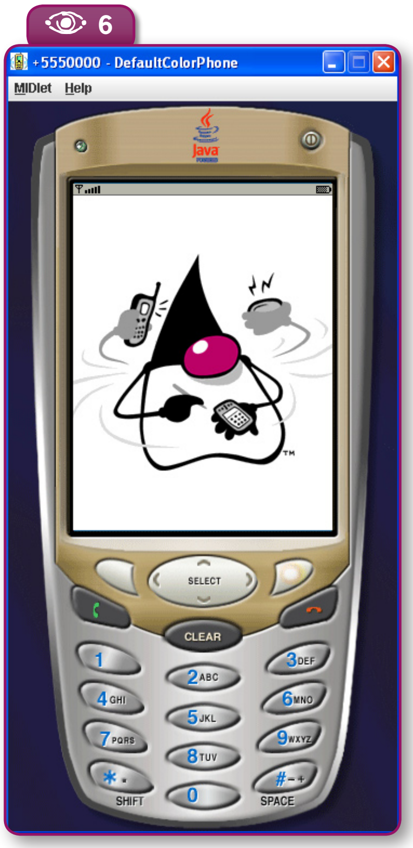


Figure 6 Splash screen in the device emulator.

Mobility Pack for CLDC/MIDP documentation index

netbeans.org/kb/55/mobility.html

7

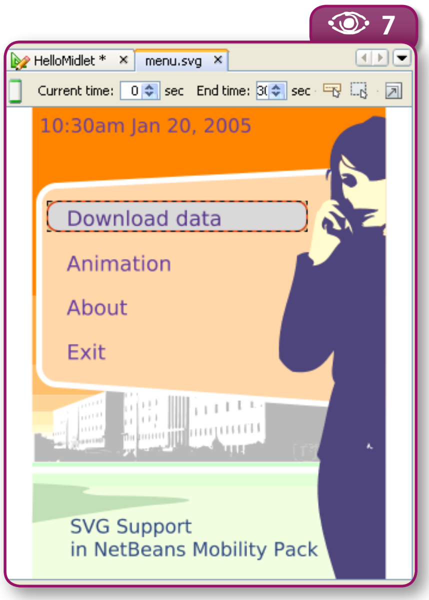


Figure 7
An interactive SVG-based menu.

the visual content. To use SVG, you'll need the Sun Java Wireless Toolkit 2.5, which is currently available as a module from NetBeans Update Center, and is bundled with the Mobility Pack starting with version 5.5.1.

In the VMD, you can add an external SVG editing tool, such as Hyperion or Ikivo, and use it to create your initial SVG graphic. Like we did in the example we just looked at, you can create a splash screen, an interactive menu, or a wait screen by dragging and dropping components into the Flow Designer. You can drop the graphic or animation on the component, and inspect the behavior of the graphic or animation as the application is run.

Figure 7 shows the VMD inspecting an animated SVG menu. You can use the VMD flow designer interface to link each menu item to a separate screen that would be called when the menu item is selected.

Reducing fragmentation with project configurations

One of the most difficult aspects of developing applications is *device fragmentation*. Mobile devices differ in a variety of attributes, such as screen size, color depth, and the proprietary or optional APIs they support. These differences often require special code or project settings for successful deployment. One solution is to create separate source code for each device you're programming to, which is almost guaranteed to be a logistics nightmare. We've already touched on the Mobility Pack solution for device fragmentation – *project configurations*.

Project configurations enable you to define the execution environment for each target device. With project configurations and code pre-processing, you can write an application and – using a single set of source code – customize, debug, and deploy a separate distribution JAR for each target device. If you need to customize your MIDlet for more devices, you add a new configuration for each device, modify the project properties, add some pre-processing code, then build and deploy the application. In most cases, you should create one configuration for each distribution JAR you plan to build for your project. For example, if you are planning to support three different screen sizes using two sets of vendor specific APIs, you should create six configurations.

Before we look at deploying our MIDlet to different devices, let's examine the three main concepts behind project configurations.

The Emulator Platform

An emulator platform simulates the execution of an application on one or more target devices. It enables you to understand the user

The Sun Java Wireless Toolkit 2.5

The Sun Java Wireless Toolkit 2.5 includes all of the advanced development features found in earlier versions, such as MIDlet signing, certificate management, integrated OTA emulation, push registry emulation, and more. New features include support for the Mobile Service Architecture (JSR-248) platform. Although this JSR does not define any new APIs, it does standardize many existing ones into a common API stack, to increase interoperability and make mobile development easier. There are also many new APIs supported, such as Security and Trust Services (JSR 177), Location (JSR 179), SIP (JSR 180), Content Handler (JSR 211), Scalable 2D Vector Graphics (JSR 226), Payment (JSR 229), and several others.

netbeans.org/kb/55/quickstart-mobilitycdc.html

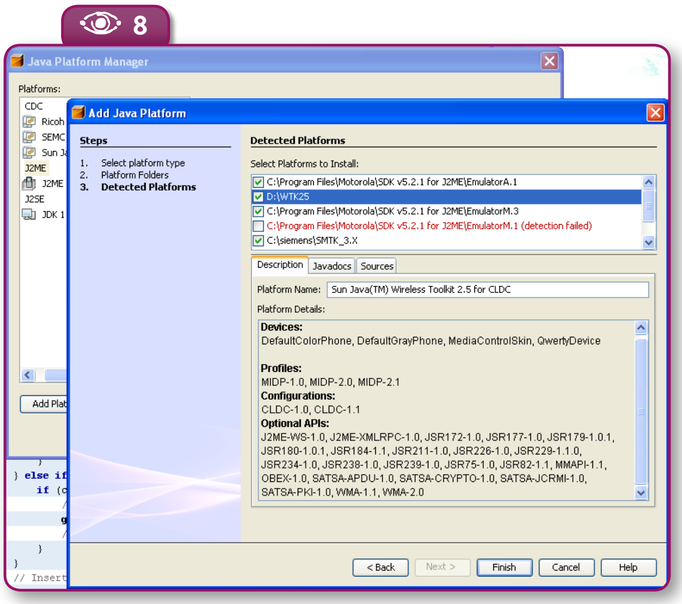
Mobility Pack for CDC Quick Start Guide

experience for an application on a particular device, and to test the portability of the application across different devices. As you have seen, the J2ME Wireless Toolkit 2.2, bundled with Mobility Pack 5.5 for CLDC/MIDP, provides several sample devices, like the DefaultColorPhone (and you can easily update to WTK 2.5).

It is important, however, to remember that an emulator can only approximate a device's performance. Environmental variables like processing speed or the strength of the wireless signal can affect performance on a real device, and should be taken into account.

A very important feature of the Mobility Pack is its ability to work with the emulators provided by major manufacturers, such as Nokia, Sony Ericsson, Siemens, and Motorola. Using the Java Platform Manager (see **Figure 8**), you can easily add any emulator that supports the Unified Emulator Interface (UEI) standards. Simply choose *Tools>Java Platform Manager*, select the J2ME Platform, and the wizard detects the emulator platforms installed on your system. Other emulators can also be added with a little more effort.

8



Steps

1. Select platform type
2. Platform Folders
3. **Detected Platforms**

Detected Platforms

Select Platforms to Install:

☑ C:\Program Files\Motorola\SDK v5.2.1 for J2ME\EmulatorA.1
☑ D:\WTK25
☑ C:\Program Files\Motorola\SDK v5.2.1 for J2ME\EmulatorM.3
☐ C:\Program Files\Motorola\SDK v5.2.1 for J2ME\EmulatorM.1 (detection failed)
☑ C:\siemens\SMTK_3.X

Platform Name: Sun Java(TM) Wireless Toolkit 2.5 for CLDC

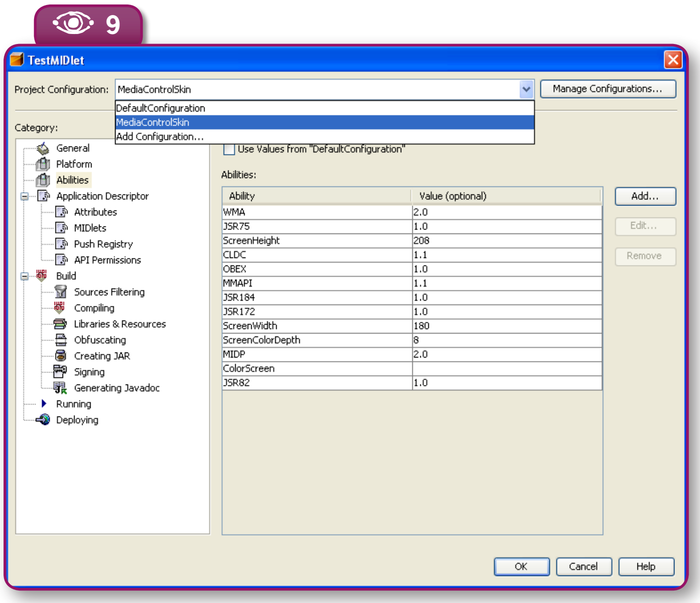
Platform Details:

Devices:
DefaultColorPhone, DefaultGrayPhone, MediaControlSkin, QwertyDevice

Profiles:
MIDP-1.0, MIDP-2.0, MIDP-2.1
Configurations:
CLDC-1.0, CLDC-1.1
Optional APIs:
J2ME-WS-1.0, J2ME-XMLRPC-1.0, JSR172-1.0, JSR177-1.0, JSR179-1.0.1, JSR180-1.0.1, JSR184-1.1, JSR211-1.0, JSR226-1.0, JSR229-1.1.0, JSR234-1.0, JSR238-1.0, JSR239-1.0, JSR75-1.0, JSR82-1.1, MMAPI-1.1, OBEX-1.0, SATSA-APDU-1.0, SATSA-CRYPTO-1.0, SATSA-JCRMI-1.0, SATSA-PKI-1.0, WMA-1.1, WMA-2.0

Figure 8
The Java Platform Manager.

9



Project Configuration: MediaControlSkin

DefaultConfiguration
MediaControlSkin
Add Configuration...

☐ Use Values from "DefaultConfiguration"

Abilities:

Ability	Value (optional)
WMA	2.0
JSR75	1.0
ScreenHeight	208
CLDC	1.1
OBEX	1.0
MMAPI	1.1
JSR184	1.0
JSR172	1.0
ScreenWidth	180
ScreenColorDepth	8
MIDP	2.0
ColorScreen	
JSR82	1.0

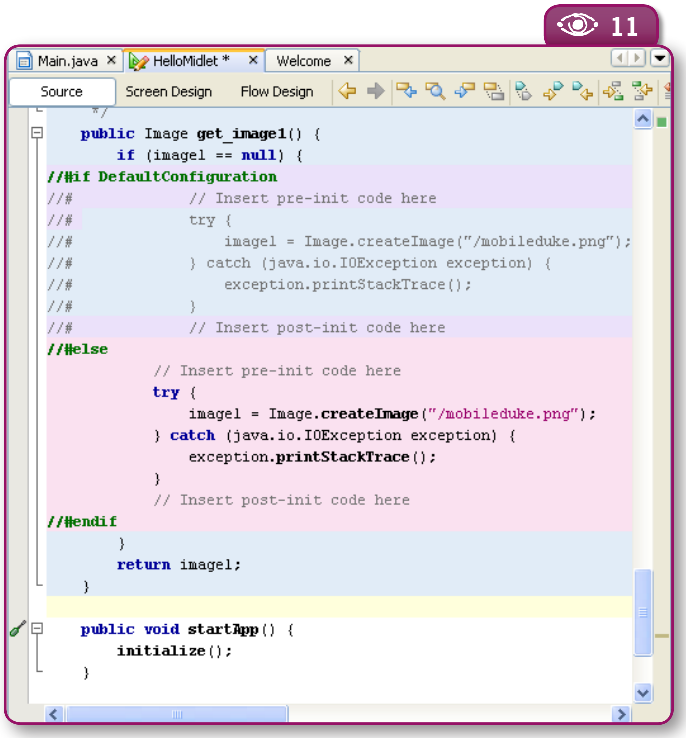
Figure 9
Abilities page.

Project properties

We can use the project properties to define many aspects of the program. As you can see in **Figure 9**, the property categories are on the left, and the properties for that category are on the right. A short list of things you can do in properties includes: defining the emulator platform, setting/checking Configuration and Profile versions and optional APIs the device supports; adding or removing the contents of the JAR and JAD files; setting the Push Registry; setting obfuscation and optimization levels; adding signing and security certificates; and setting deployment options. The Abilities shown in **Figure 9** list attributes that might be shared by different devices, and therefore might be shared when you are adding pre-processing code.

Let's take a look at this using the two configurations we've created for our **TestMIDlet**. The first is the DefaultConfiguration we've been using so far. The second is the MediaControlSkin that you selected when first creating the project. Before we discuss project properties and pre-processing code, let's take a quick look at the MediaControlSkin device emulator.

To switch configurations, choose the



```
    */
    public Image get_image1() {
        if (image1 == null) {
//#if DefaultConfiguration
//#                // Insert pre-init code here
//#                try {
//#                    image1 = Image.createImage("/mobileduke.png");
//#                } catch (java.io.IOException exception) {
//#                    exception.printStackTrace();
//#                }
//#                // Insert post-init code here
//#else
            // Insert pre-init code here
            try {
                image1 = Image.createImage("/mobileduke.png");
            } catch (java.io.IOException exception) {
                exception.printStackTrace();
            }
            // Insert post-init code here
//#endif
        }
        return image1;
    }

    public void startApp() {
        initialize();
    }
```

Figure 11
Preprocessor code.

MediaControlSkin from the Configuration drop-down menu in the IDE toolbar. Then choose *Run>Run Main Project*. It's the same **TestMIDlet** as before, but this time the device emulator and its display are thinner – so we lose a little bit of the splash screen, as shown in **Figure 10**.

Preprocessing code

Preprocessing modifies the code in your source files before the code is parsed by the compiler. The preprocessor modifies the code according to preprocessor directives you insert into the code as code blocks with beginning and ending directives. These code blocks are marked visually in the Source Editor and are included (or excluded) when you build the JAR for a specific project configuration or ability. You can use these code blocks to create, manage, and track code that is specific to one or more project configurations or abilities.

Now that we've defined the key concepts of project configurations, let's go back to our example and say now that we want to use a different graphic for the MediaControlSkin device. We already have our two project configurations defined, so what we need to do is add some pre-processing code so that the compiler knows it should use one image for DefaultConfiguration and another for MediaControlSkin.

Click the *Source* button to view the **TestMIDlet** source code. Then scroll down to the **get_image1()** method. Notice that some sections have a blue background – these are "guarded blocks" that are generated by the VMD, and cannot be edited. Highlight the code beginning with "//Insert pre-init code here and ending with "//Insert post-init code here." Right click on the selection and choose *Preprocessor Blocks>Create If/Else Block*. A menu with all the available configurations and abilities appears. Double click on DefaultConfiguration. Your code should look like **Figure 11**.

Notice that the //**#else** directive has a pink background. This code *can* be edited. Change the graphic name "/mobileduke.png" to the name of another graphic (such as "/veryproudduke.

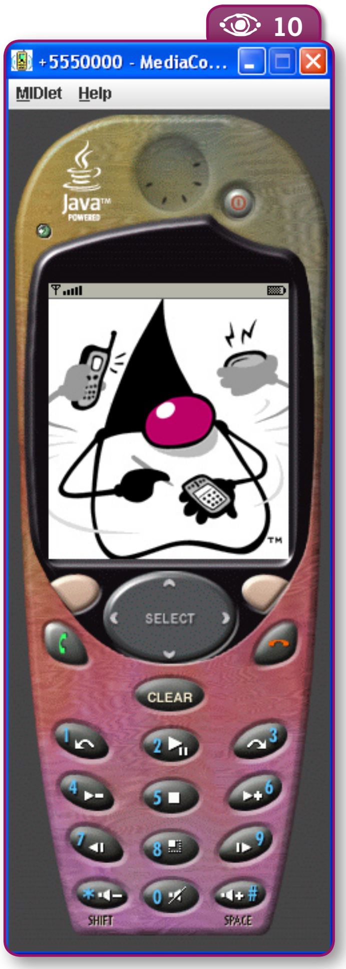


Figure 10
The MediaControlSkin Emulator.

netbeans.org/kb/55/mobilitycdc.html

Mobility Pack for CDC documentation index png"). Now, when you run our MIDlet using the DefaultConfiguration, the emulator will display the "mobile duke" image. When you run the MIDlet using the MediaControlSkin configuration, or any other configuration you add later, the emulator will display the second graphic. This is a very simple example of what's possible with preprocessor blocks, but it hopefully gives you a taste of what they can do.

Deploying to multiple devices

Now that you have an application that works with two devices, it's time to deploy it. The deployment property page shown in **Figure 12** shows the different deployment methods available.

Because deployment is set in the project Properties, you can define a different deployment for each configuration. When you have chosen your deployment method, choose *Build>Build All Main Project Configurations*. Then you'll have a JAR for each target device you're programming for.

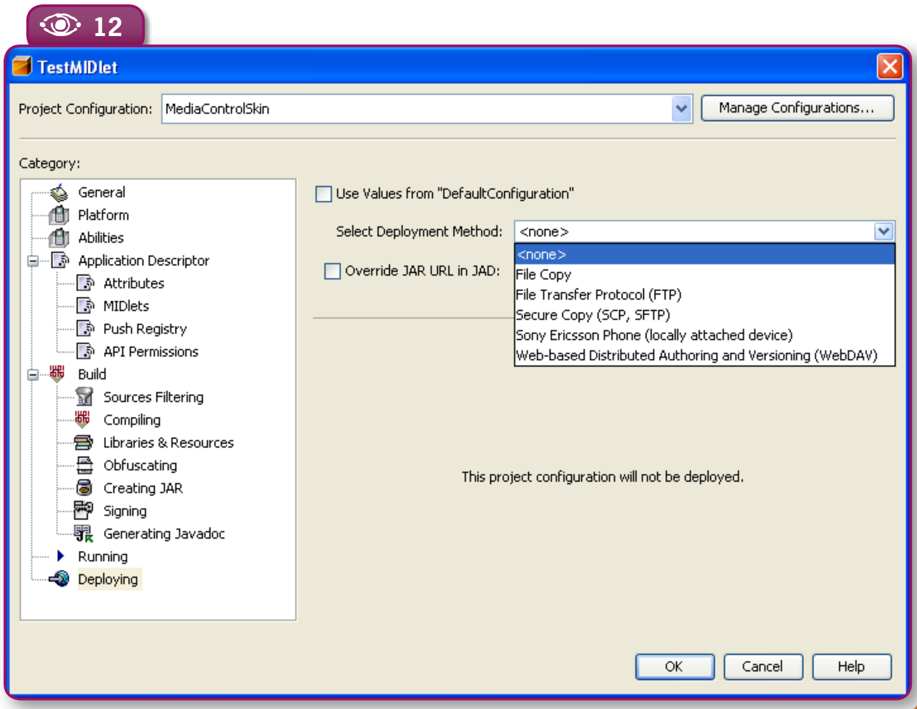
Other features

Our example was purposefully kept simple to focus on the design, configuration, and deployment features of the Mobility Pack for CLDC. But there are two other important features we'd like to mention quickly before we move on to the CDC Pack.

The End-to-End Bridge technology is a set of two wizards that enable you to quickly modify your MIDlet to consume Web Services. The J2ME Web Service Client creates a client-side proxy that connects directly to Web Services that support the JSR-172 (J2ME Web Services) specification. The Mobile Client to Web Application Generator generates a servlet that connects to a web application that includes a Web Service client.

Another important feature is JMUnit testing support. The Mobilty Pack for CLDC provides built-in JMUnit support for generating and executing unit tests for MIDP/CLDC applications. You can generate and navigate to tests by selecting any class or package node in the Projects window and choosing from the *Tools>JUnit* menu.

12



Project Configuration: MediaControlSkin

Category:

☐ Use Values from "DefaultConfiguration"

Select Deployment Method: <none>

☐ Override JAR URL in JAD:

- <none>
- File Copy
- File Transfer Protocol (FTP)
- Secure Copy (SCP, SFTP)
- Sony Ericsson Phone (locally attached device)
- Web-based Distributed Authoring and Versioning (WebDAV)

This project configuration will not be deployed.

Figure 12 Deployment properties.

NetBeans Mobility Pack for CDC

The Mobility Pack for CDC (Connected Device Configuration) makes it possible to create, test, and deploy applications for several CDC platforms including the Sun Java CDC Toolkit, Sony Ericsson CDC Platform 1, Nokia S80, SavaJe and Ricoh MFP, as well as Windows CE using NSIcom's CrEme CDC virtual machine. Although it is not yet as complete as the MIDP/CLDC Pack, many new features will be added when Mobility Pack 6.0 is released.

You create CDC projects in the same manner as MIDP/CLDC projects, using the New Project Wizard. Before you begin, you will want to install the Sun Java Toolkit for CDC or an emulator platform from one of the growing list of manufacturers the Mobility Pack supports. You can find this list in the NetBeans Mobility Pack for CDC Quick Start Guide.

To get started on a project, Choose *File>New Project*. Choose the category *CDC*, project type *CDC Application*. The wizard will guide you through the rest of the steps for creating your Main Project.

Once you've created a project, you can use the Matisse GUI Builder with either the AGUI toolkit or the Personal Profile 1.0, in the same way you would use it for regular Java SE development. For Personal Profile GUI development, right click the Main.java form in the GUI Builder, and choose *Set Layout>Free Layout*. Then drag and drop components from the Palette window into the Design Area of the GUI Builder. You can also take advantage of JUnit testing and other key features of the NetBeans IDE when developing CDC applications in the Mobility Pack for CDC.

What's coming in NetBeans Mobility Pack 6.0

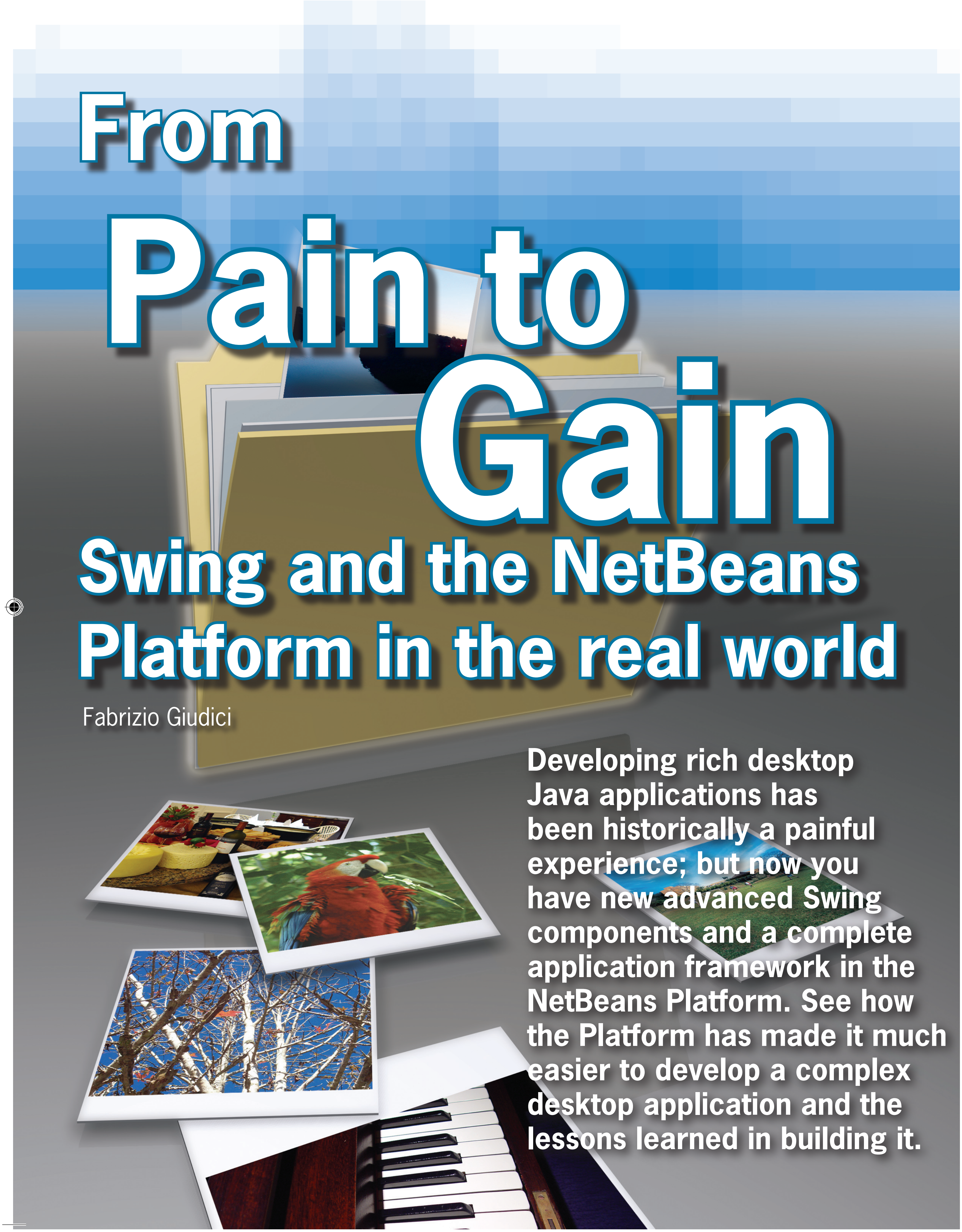
The NetBeans Mobility Pack has many dramatic changes coming up. One of the most significant changes, as we've mentioned before, is that the CLDC/MIDP and CDC Packs will be merged into a single UI, making it easier to create end-to-end applications. Other exciting new features include:

- New custom components to simplify programming, including a File Browser, an SMS Composer, a Login Screen and a Personal Information Manager (PIM) Browser
- VMD support for the MIDP 2.x Game API that allows creating tiled and animated layers for environment design, and support for animated character and sprites.
- Improved VMD UI, including support for non-visual components and a design analyzer.
- CDC support for project configurations and pre-processing blocks.

Conclusions

This article was intended to give you a hands-on sense of the capabilities of the Mobility Pack for CLDC/MIDP and CDC, and a running start on building your first mobile application. You can learn more about the Mobility Pack by reading the tutorials and articles on the NetBeans website, joining the NetBeans community of developers, and most importantly, by going out there and creating great mobile applications!

Anatole Wilson (*anatole.wilson@sun.com*) lives in Pittsburgh, Pennsylvania and has been the Senior Technical Writer for the Mobility Pack since its inception. He has worked for various high-tech companies, including IBM and Oracle, and, as a freelance writer, has written articles for publication in various magazines.



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-

bluemarine.tidalwave.it
The blueMarine Project

aerith.dev.java.net
Aerith

blogs.sun.com/geertjan
Geertjan Wielenga's blog

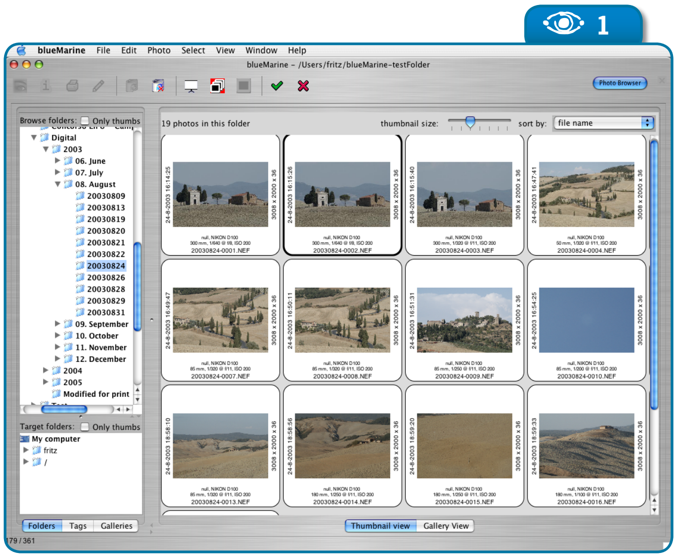


Figure 1 The old blueMarine main window, developed on plain Swing.

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.

Romain Guy's blog

www.curious-creature.org

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

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

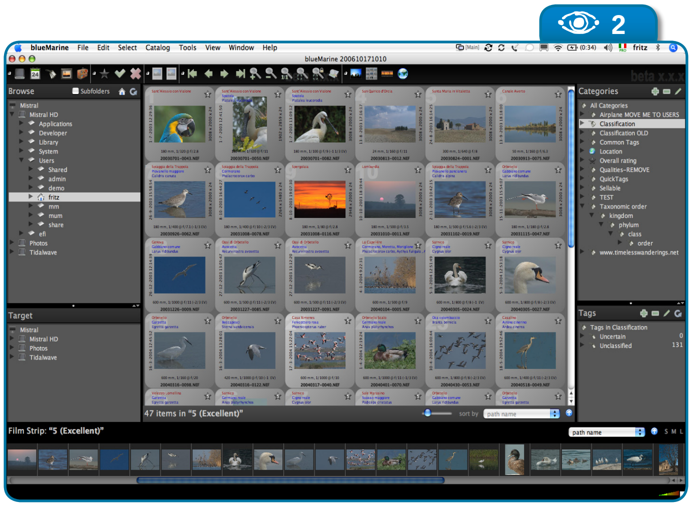


Figure 2
The main window of the new blueMarine application, developed on the NetBeans Platform.

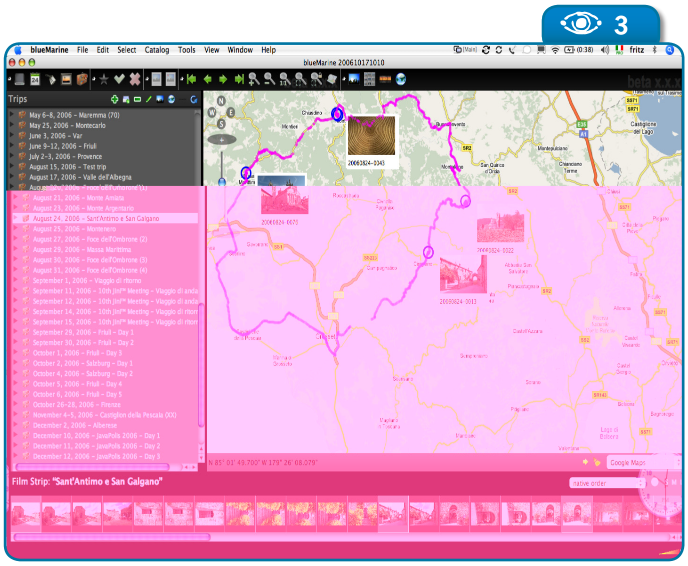


Figure 3
The Map Viewer with geo-tagged photos uses the *JXMapViewer* component.

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.

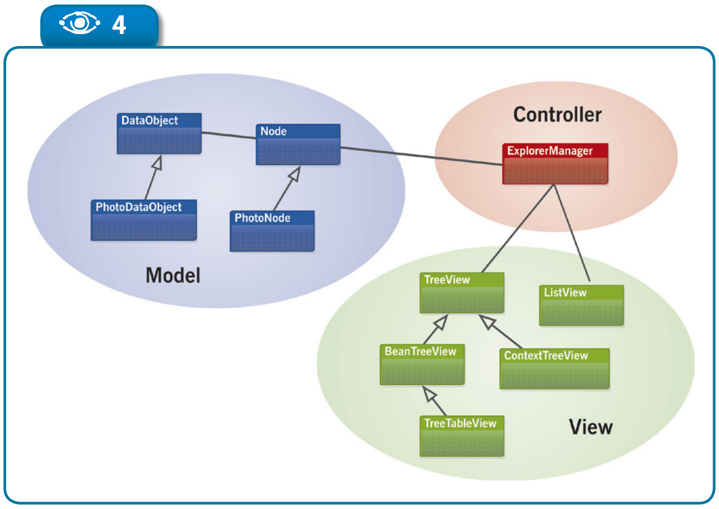


Figure 4 NetBeans MVC components.

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.

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 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.

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.

5

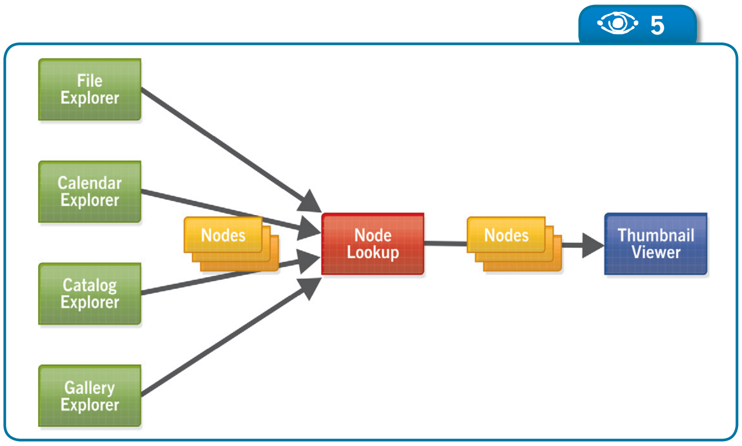


Figure 5 The role of *Node* objects for inter-module communication.

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

swinglabs.org SwingLabs

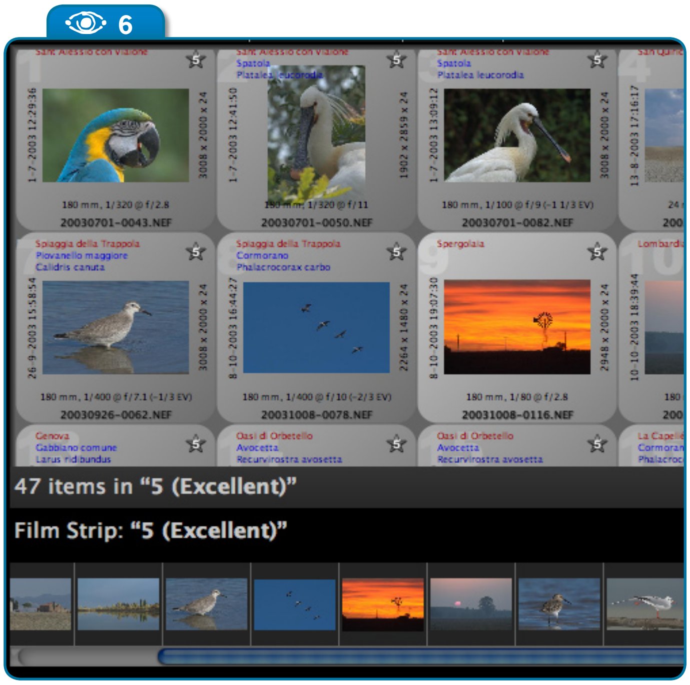


Figure 6 The Thumbnail Viewer and the Film Strip Viewer must show the same nodes – with the same selection too.

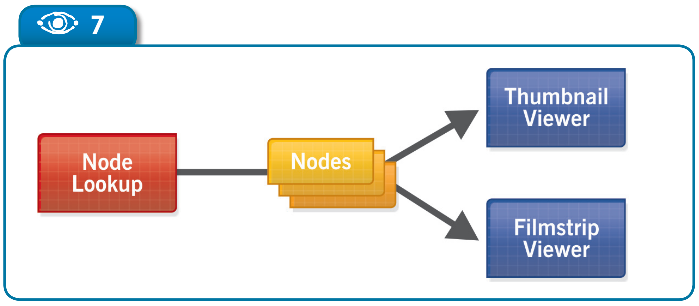


Figure 7 Multiple, synchronized views are implemented by just listening to the same *Node*'s changes.

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 filesystem 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.

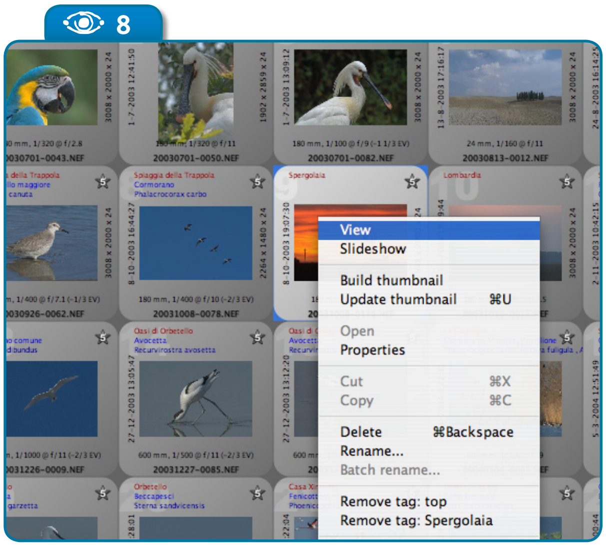


Figure 8
Each thumbnail is a rendered *Node* object bound to a *PhotoDataObject* and can pop up a menu with contextual actions. The grid arrangement and the look of thumbnails are implemented by means of a customized *ListView*.

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-**

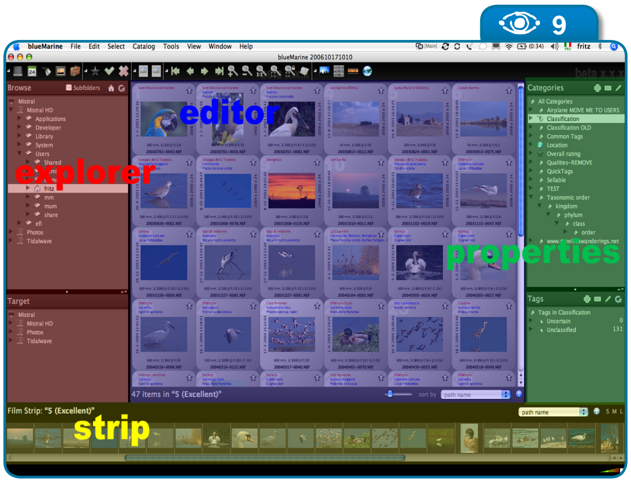


Figure 9
Modes in the main window of blueMarine – "strip" is a custom mode; each mode contains a *TopComponent*.

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);
```

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();
}
```

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

The author's blogs

weblogs.java.net/blog/fabriziogiudici
www.tidalwave.it/blog

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.

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

The Quaqua look and feel

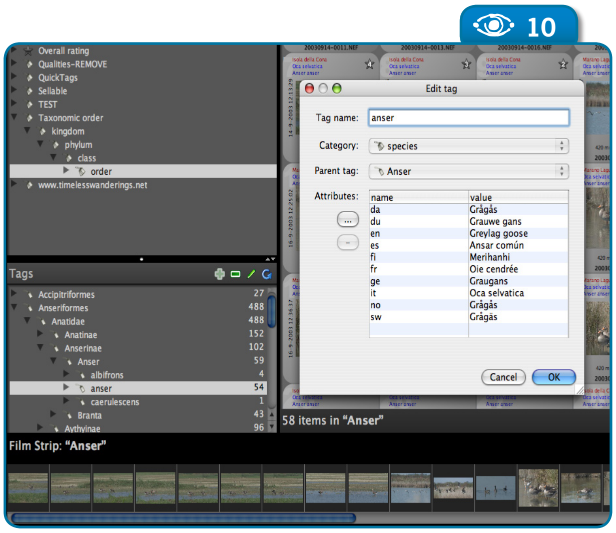
quaqua.dev.java.net

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;
  }
}
```

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).



Edit tag

Tag name: anser

Category: species

Parent tag: Anser

Attributes:

name	value
da	Grågås
du	Grauwe gans
en	Greylag goose
es	Ansar común
fi	Merihanhi
fr	Oie cendrée
ge	Graugans
it	Oca selvatica
no	Grågås
sw	Grågäs

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.

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;
  }
...
```

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

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.

OpenOf NetBeans Integration

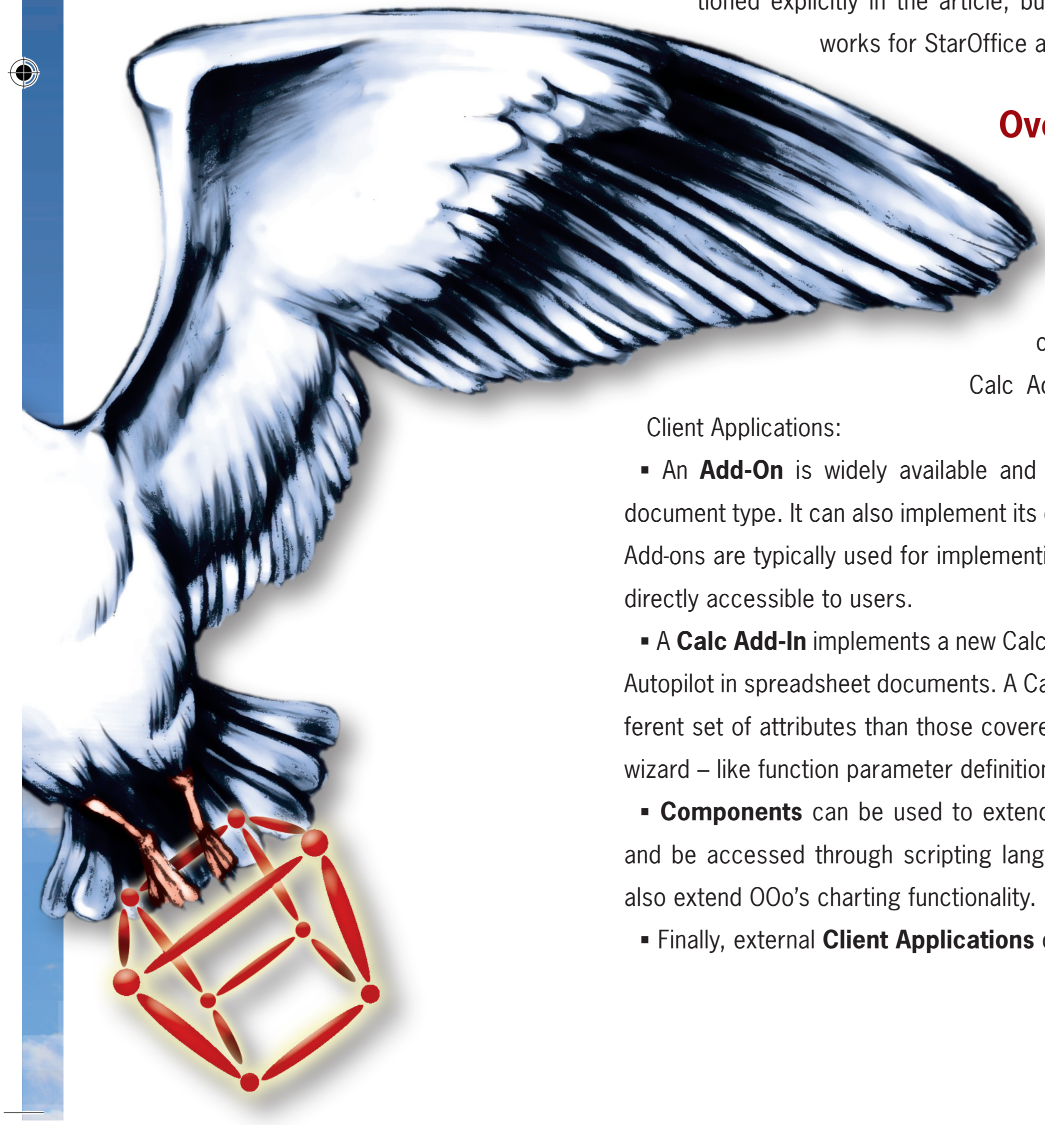
Kay Koll

Using a new OpenOffice.org plug-in module to create, build, deploy and debug OpenOffice.org/StarOffice extensions and client applications

fice.org

Automating office tasks, implementing document-based workflows or building corporate solutions are common tasks that can benefit from using the API of an Office suite. OpenOffice.org with its millions of users, and Java with its huge community have several similarities; both, for example, are multiplatform and open source. But a tool to combine both worlds was missing. That's where the new OpenOffice.org plugin module for NetBeans comes into the game.

In this article we will show you how to use the new module to build Java components which extend OpenOffice.org's general functionality; we also create a new Calc function and a client application that accesses OOo features. (StarOffice won't be mentioned explicitly in the article, but everything shown here works for StarOffice as well.)



Overview

The OpenOffice.org plugin module for NetBeans provides four wizards which let you create general Add-Ons, Calc Add-ins, Components and Client Applications:

- An **Add-On** is widely available and not limited to a certain document type. It can also implement its own toolbars and menus. Add-ons are typically used for implementing new features that are directly accessible to users.
- A **Calc Add-In** implements a new Calc function for the Function Autopilot in spreadsheet documents. A Calc function requires a different set of attributes than those covered by the general Add-On wizard – like function parameter definitions.
- **Components** can be used to extend the OpenOffice.org API and be accessed through scripting languages. Components can also extend OOo's charting functionality.
- Finally, external **Client Applications** can use OpenOffice.org's

wiki.services.openoffice.org/wiki/OpenOffice_NetBeans_Integration

The project's homepage on the OpenOffice.org Wiki functionality to create, convert, print or manipulate documents. Thus, OpenOffice.org can be used as powerful rendering and printing engine within a larger solution.

The new wizards create NetBeans projects with all necessary configurations, such as links to the OpenOffice.org Java libraries. They also define build targets, perform remote debugging setup and generate Java code skeletons, among other operations.

OpenOffice.org Developers Guide

api.openoffice.org/DevelopersGuide/DevelopersGuide.html

Requirements, installation and configuration

To use the NetBeans OOo plugin module you need NetBeans 5.5 or newer, and either OpenOffice.org 2.0.4 or StarOffice 8 PU4; also needed is the OpenOffice.org SDK 2.0.4 or newer.

The wizards are provided in a common NetBeans module that can be installed and updated via the Update Center. Just download the file *api.openoffice.org/Projects/NetBeansIntegration/org-openoffice-extensions.nbm* and use *Run>Update Center>Install Manually Downloaded Modules (.nbm Files)*[1].

Setup is straightforward. There are just two configuration items available: the paths to the OpenOffice.org installation and to the SDK. These are accessible during installation and also via the *Tools >Options>Miscellaneous* dialog.

Creating OOo extensions

All four extension types are packed as OOo packages. This package format was introduced in OpenOffice.org 2.0.4 and uses the file extension *.OXT*. A corresponding MIME type is registered in OpenOffice.org, which enables users to install extensions simply by double clicking.

Add-Ons

The *File>New Project* dialog provides a new *OpenOffice.org* category that opens the Add-On wizard. Start the wizard, and enter "myAddOn" for both the project and add-on names. Set the Java package to "org.openoffice"; provide the project folder, and check the *Create Menu* and *Create Toolbar* checkboxes.

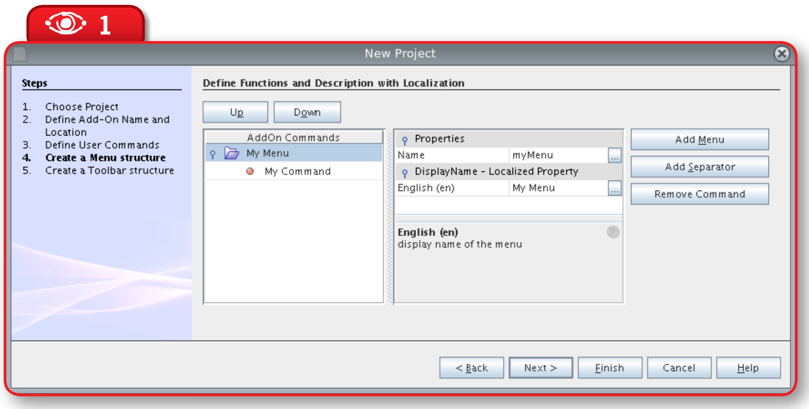
Click *Next* and you'll be able to specify the add-on commands. Each command can have an icon assigned to it on the toolbar. Ideally, you should provide four different icons (two different sizes in two contrast levels), but it's possible to use the same image for all icon types. If you do this OpenOffice.org will scale the image accordingly.

You also need to enter a display name for the command. This name is used for the menu and toolbars and can be different from the command name. Note that only the display name can be translated: the wizard lets you define different locales, but the command name is the same for all locales.

In the next step you define the menu structure. This is optional, as not all add-ons have their own menus. **Figure 1** shows the menu definition wizard page.

The next step is also optional: the definition of a toolbar to call the add-on's commands (see **Figure 2**). The icons in the toolbar preview should look familiar, as they were specified in step three. Set the names as "myAddOn", define the icons and set the category.

After you click *Finish*, the wizard creates two configuration files and a Java class



1

New Project

Steps

1. Choose Project
2. Define Add-On Name and Location
3. Define User Commands
4. **Create a Menu structure**
5. Create a Toolbar structure

Define Functions and Description with Localization

Properties

Name: myMenu

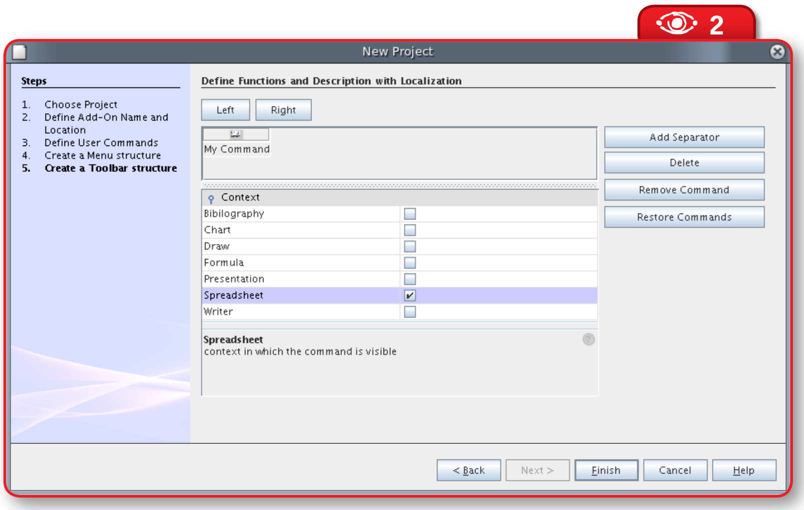
DisplayName - Localized Property

English (en): My Menu

English (en)
display name of the menu

Figure 1
Defining the add-on's menu structure.

[1] By the time you read this, it's possible that the module will show up in the Update Center, making the manual download unnecessary.



Steps

1. Choose Project
2. Define Add-On Name and Location
3. Define User Commands
4. Create a Menu structure
5. **Create a Toolbar structure**

Define Functions and Description with Localization

My Command

Context

Bibilography	☐
Chart	☐
Draw	☐
Formula	☐
Presentation	☐
Spreadsheet	☑
Writer	☐

Spreadsheet
context in which the command is visible

Figure 2
Defining the toolbar for the add-on.

skeleton. The *AddOns.xcu* configuration file includes the add-on parameters, and *ProtocolHandler.xcu* defines the protocol handler configuration. Protocol handlers are part of OpenOffice.org dispatch framework; they bind user-interface controls, such as menu or toolbar items, to the functionality of OpenOffice.org. Everything reachable through the user interface is described by a command URL and corresponding parameters.

The structure of the *ProtocolHandler.xcu* file defines a namespace for the add-on (*org.openoffice.myaddon*, for our example). All commands defined by the same add-on use this namespace. See **Listing 1**.

The Java code skeleton looks more complicated than it really is. Most of the methods are necessary only for OpenOffice.org internal implementation reasons and don't need to be changed at all.

OpenOffice.org calls the **dispatch()** method each time the protocol handler routes commands. The fully implemented **dispatch()** method, which adds the "Hello World" message to the current document is shown in **Listing 2**. Note that only the lines in bold are new; the rest were part of the skeleton. The code basically determines the current document and then adds text to it. Notice also that the **aURL** parameter uses the OpenOffice.org specific **com.sun.star.util.URL** class instead of the **java.net.URL** class.

Since all commands fired by the user interface are passed through the **dispatch()** method, it's necessary to filter explicitly for the namespace *org.openoffice.myaddon*, which represents the commands of our add-on. Finally, **myAddOn** is the command that's fired when the user calls the add-on via the toolbar or menu.

Our add-on is now ready to deploy. Right click the project name and choose *Deploy Office Extension* from the context menu. NetBeans compiles all necessary files, creates an OpenOffice.org extension package file, and deploys it. Depending on the setup of the add-on, a new top-level menu and/or a toolbar are displayed in

Listing 1. Excerpt from the ProtocolHandler.xcu configuration file.

```
<node oor:name="HandlerSet">
  <node oor:name="org.openoffice.myAddOn" oor:op="replace">
    <prop oor:name="Protocols" oor:type="oor:string-list">
      <value>org.openoffice.myaddon:*</value>
    </prop>
  </node>
</node>
```

Listing 2. dispatch() method example.

```
public void dispatch( URL aURL, PropertyValue[] aArguments )
{
  if ( aURL.Protocol.compareTo("org.openoffice.myaddon:") == 0 ) {
    if ( aURL.Path.compareTo("myAddOn") == 0 ) {
      XTextDocument xDoc = (XTextDocument) UnoRuntime.queryInterface(
           XTextDocument.class, m_xFrame.getController().getModel());
      xDoc.getText().setString("Hello World");
      return;
    }
  }
}
```

OpenOffice.org. For our example both should show up (see the new menu in **Figure 3**).

Developing more complex add-ons will probably require an extensive debugging session. However, as the add-ons run in OpenOffice.org's JVM, NetBeans' built-in debugger won't work. We need remote debugging.

The OOo plugin module adds a command to the project's context menu for starting a remote debugging session, which means that a manual setup of the client JVM or the remote debugger is not required. To use this feature, set a breakpoint within the **dispatch()** method, then call *Debug Extension in Target Office* as displayed in **Figure 4**. An OpenOffice.org instance will start automatically. Choose *My Command* from the add-on's menu and the debugger will stop at the breakpoint.

Calc Add-Ins

Let's now see how to create a Calc Add-In extension, which implements OpenOffice.org Calc functions. These functions are tightly integrated with the Calc application, so users will not recognize the differences between a standard function and one provided by an add-in; there are no new menus, toolbars or other evidences of an extension.

The Calc Add-In wizard is also located in the *StarOffice/OpenOffice.org* category of the *File>New Project* dialog. Start the wizard and enter a name and location for the NetBeans project. We'll use "myAddIn" for both names and "org.openoffice" again as the package name; also make sure *Create backward compatible Calc Add-In* is unchecked. Click *Next* to enter the name and parameters of the Calc functions.

The definition of a Calc Add-In function requires specifying the following parameters:

- The name of the Java method which implements the Calc function.
- The data type of the result of the new function.
- The exception the Java implementation throws in case of errors. An additional dialog provides access to all available exceptions. (This property is optional.)
- The Category where the function is listed within Calc's Function Wizard.
- The function's display name. This can be different from the name of the corresponding Java method.

Figure 5 shows how the wizard presents these parameters.

There are some additional parameters, which are all localizable. Calc functions are usually localized, having different names for each language; for example, the function **Sum()** in the English-language Calc is

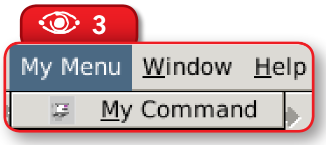


Figure 3
The new add-on menu.

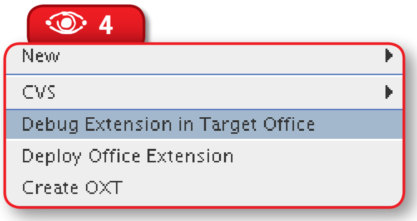
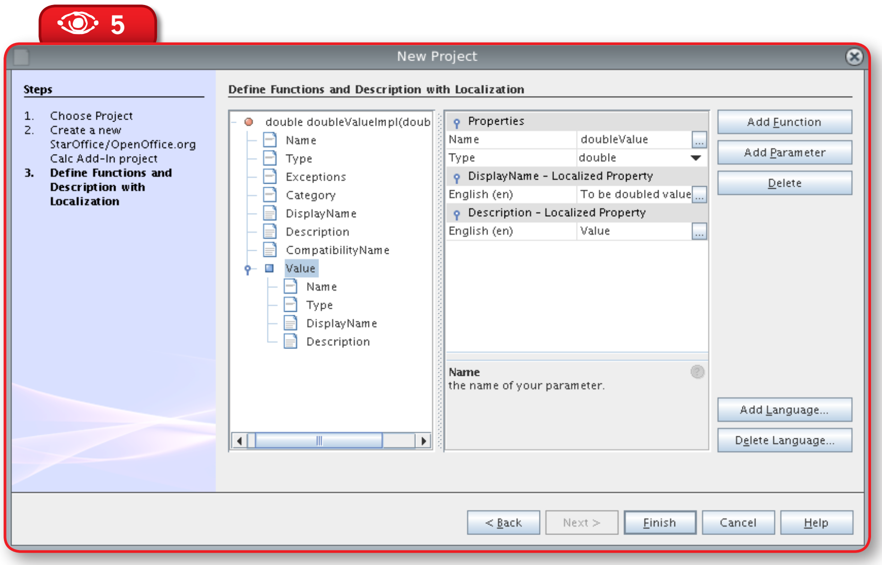


Figure 4
Remote debugging context menu item.



Steps

1. Choose Project
2. Create a new StarOffice/OpenOffice.org Calc Add-In project
3. **Define Functions and Description with Localization**

Define Functions and Description with Localization

Properties	
Name	doubleValue
Type	double
DisplayName - Localized Property	
English (en)	To be doubled value
Description - Localized Property	
English (en)	Value

Name
the name of your parameter.

Figure 5
Defining a function and its parameters in the Calc Add-In wizard.

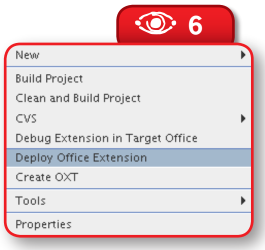
named **Summe()** in German Calc releases.

The *Description* parameter indicates the purpose of the function, and is displayed in Calc's Function Wizard. *Compatibility Name* is necessary to deal with Microsoft Excel integration; it's optional and usually not necessary. Finally we have the optional parameters of the Calc function. They require a specification of the data type, the implementation name, and the displayed name and description.

To create our simple Calc Add-In, set Name to "doubleValueImpl" and Type to *double*; leave the Exceptions section empty; set Category to "Add-In" and Displayed Name to "doubleValue". Change Displayed Description to "Simple Calc Add-In: Doubles the given value", and set Compatibility Name to "doubleValue". Then provide the following values for the first parameter:

- Name – "doubleValue"
- Type – double
- Displayed Name – "Value"
- Displayed Description – "Value to be doubled"

Click *Finish* and several add-in-related files will be created. The most important is the Java class for the add-in implementation. There's also the configuration file *CalcAddin.xcu*, which holds the add-in's parameters. Functions exported by the add-in need to be defined in a new interface. The function names in this interface, together with the add-in's service name, are used internally to identify an add-in function. The *myAdd-In.idl* and *XmyAddIn.idl* files define this service and the interface. They are used by tools and compilers available in the OpenOffice.org SDK, which build Java source and header files; but this process is hidden by the OOo plugin module.



New
Build Project
Clean and Build Project
CVS
Debug Extension in Target Office
Deploy Office Extension
Create OXT
Tools
Properties

Figure 6
Deployment context menu

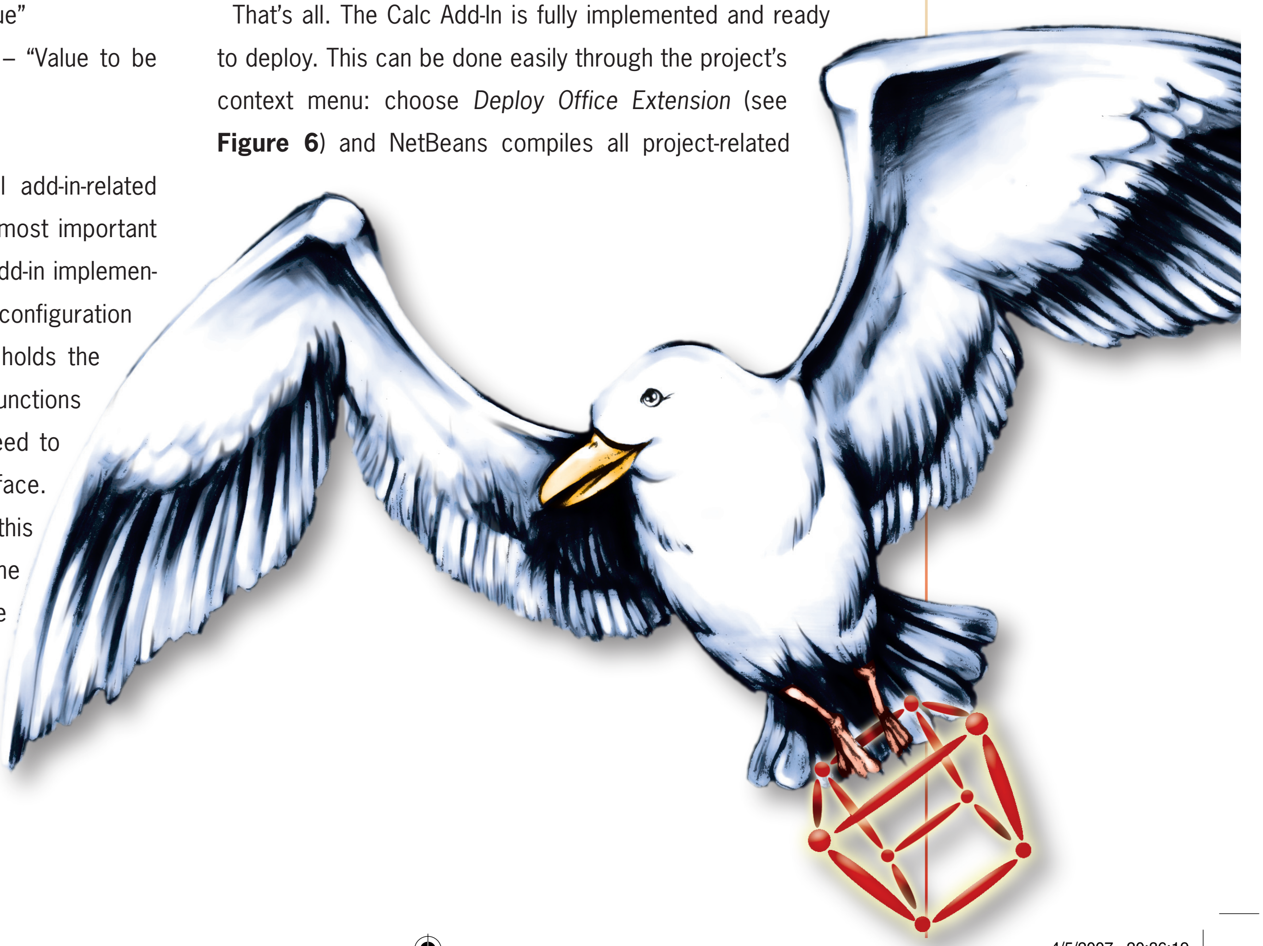
Most of the initial skeleton code need not be changed. Our **doubleValueImpl()** method is called by the add-in and provides the implementation of its functionality. The implementation is really simple; it just doubles all values given by the user:

```
public double doubleValueImpl(double doubleValue)  {
  return doubleValue * 2;
 }
```

NetBeans will complain that the **XmyAddIn** interface is missing, which is true so far. The reason is that the interface is defined in a UNO IDL (Interface Definition Language) file and not as a Java class. The plugin will create Java code based on this IDL file as well as other add-in related services and interfaces automatically when the project is compiled.

UNO (Universal Network Objects) is OpenOffice.org's component technology.

That's all. The Calc Add-In is fully implemented and ready to deploy. This can be done easily through the project's context menu: choose *Deploy Office Extension* (see **Figure 6**) and NetBeans compiles all project-related



files, builds an OXT extension package and installs it in OpenOffice.org.

You can test the new add-in by starting a Calc instance and creating a new spreadsheet document. Then call the Calc's Function wizard, where you'll see the new function listed under the Add-In category. To verify that our new function works, enter in any cell the formula "**=doublevalue(3)**". As expected, Calc will produce 6 as the result.

Download page of the latest release of the OpenOffice.org SDK

download.openoffice.org/sdk.html

Client Applications

A client application is an external solution that makes use of the OpenOffice.org functionality instead of extending it. It can use OOo to convert or process any document supported by the office suite.

The Client Application wizard is also available in a OpenOffice.org New Project category. You just need to start the wizard and enter the name and location of the NetBeans project; no further settings are required. Click *Finish*, and the wizard creates the Java skeleton.

As a client application is not an integrated part of OpenOffice.org, most of the configuration and IDL files are not necessary. The NetBeans project consists of just a Java class and classpath settings for OpenOffice.org Java libraries.

You'll notice that the code skeleton for a client application is quite small in comparison to the add-in and add-on skeletons. It is basically not much different from the Java class code generated by the general Java Class wizard. The implementation of the **main()** method contains a single line of code (besides exception handling):

```
XComponentContext xContext = Bootstrap.bootstrap();
```

The generated class works as a client of an OpenOffice.org process, with OpenOffice.org acting as a server with its own component context. The client program initializes the Universal Network Objects technology (UNO) and gets the component context from the OOo process. This initialization process establishes a pipe connection to a running OpenOffice.org process (starting a new process if necessary) and returns the remote component context.

The **getServiceManager()** method from the component context obtains the remote service manager from the OpenOffice.org process, which allows access to the complete office functionality available through the API:

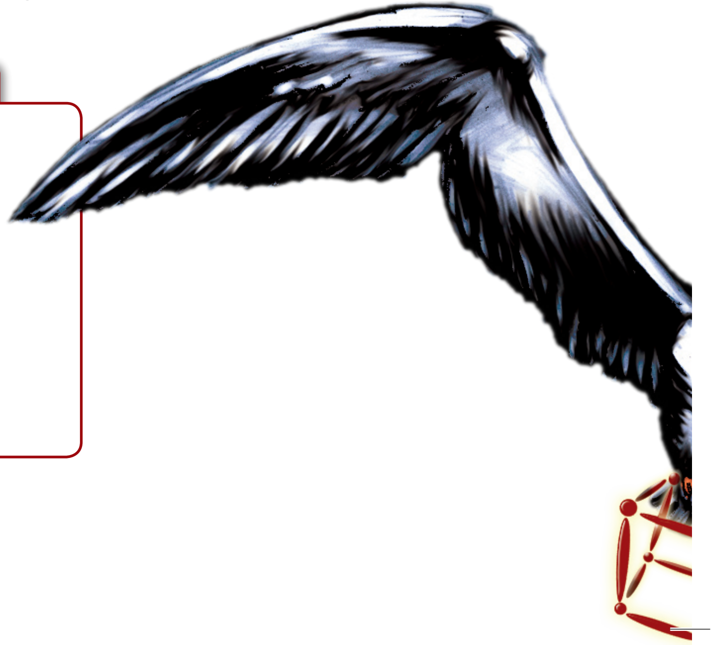
```
XMultiComponentFactory xMCF =
      xContext.getServiceManager();
```

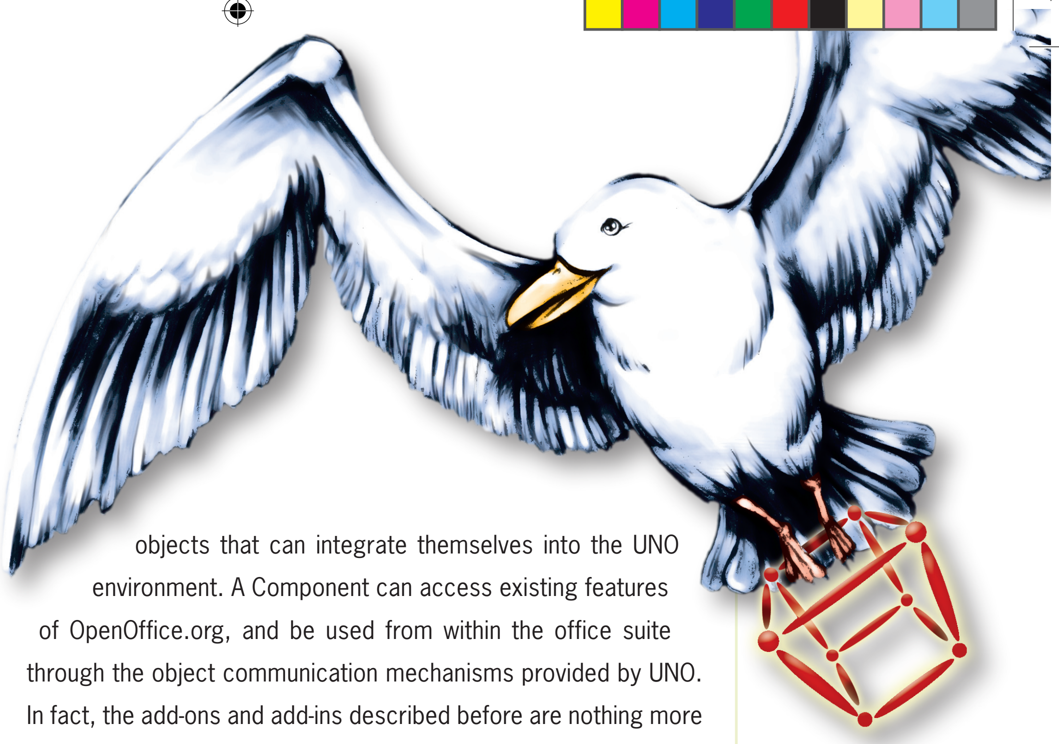
Having the service manager, we can obtain the OpenOffice.org Desktop, which handles application windows and lets you load and create documents. The **com.sun.star.frame.Desktop** service represents this Desktop:

```
XDesktop xDesktop = (XDesktop) UnoRuntime.
queryInterface(XDesktop.class,
    xMCF.createInstanceWithContext(
          "com.sun.star.frame.Desktop",xContext));
```

Listing 3. Creating a text document.

```
XComponentLoader xComponentLoader =
   (XComponentLoader) UnoRuntime.queryInterface(
      XComponentLoader.class, xDesktop);

PropertyValue xEmptyArgs[] = new PropertyValue[0];

XComponent xComponent =
   xComponentLoader.loadComponentFromURL(
      "private:factory/swriter",
      "_blank", 0, xEmptyArgs);

XTextDocument xTextDocument =
  (XTextDocument) UnoRuntime.queryInterface(
      XTextDocument.class, xComponent);
```





Now we have an instance of the Desktop without a document; but a text document is necessary to display our greeting. The **XComponentLoader** interface exports the **loadComponentFromURL()** method to load and create a document. See it in use in **Listing 3**. The *private:factory/swriter* URL creates a new text document.

The new document will show a cursor waiting for input. This input has to come from the client application. The OOo API uses a text cursor abstraction to add text to the document, represented by the **XTextCursor** interface:

```
XText xText = xTextDocument.getText();
XTextCursor xTextCursor =
   (XTextCursor) xText.createTextCursor();
```

The blinking cursor in the document and **xTextCursor** are independent of each other. OpenOffice.org Writer uses MVC to separate the content/model from the view. The text cursor is the view in this context, and the cursors created by **createTextCursor()** are the model. You can create several models for text cursors.

Finally, the method **insertString()** adds the message to the document:

```
xText.insertString( xTextCursor, "Hello World", false );
```

Components

OpenOffice.org can be extended by Components. These are shared libraries or JAR files with the ability to instantiate objects that can integrate themselves into the UNO environment. A Component can access existing features of OpenOffice.org, and be used from within the office suite through the object communication mechanisms provided by UNO. In fact, the add-ons and add-ins described before are nothing more than specialized UNO components.

Components created by the Component wizard do not require access to a menu or to toolbars, nor do they extend the Calc function repository. They can be used to implement new interfaces and services. This flexibility and power makes it impossible to create a simple "Hello" component, and creating a fully working Component would go beyond the scope of this article. There are many excellent articles and documents available which describe the creation of new OpenOffice.org interfaces and services. Specifically, we refer you to Chapter 4 of the OpenOffice.org Developer Guide, which is a good source of examples.

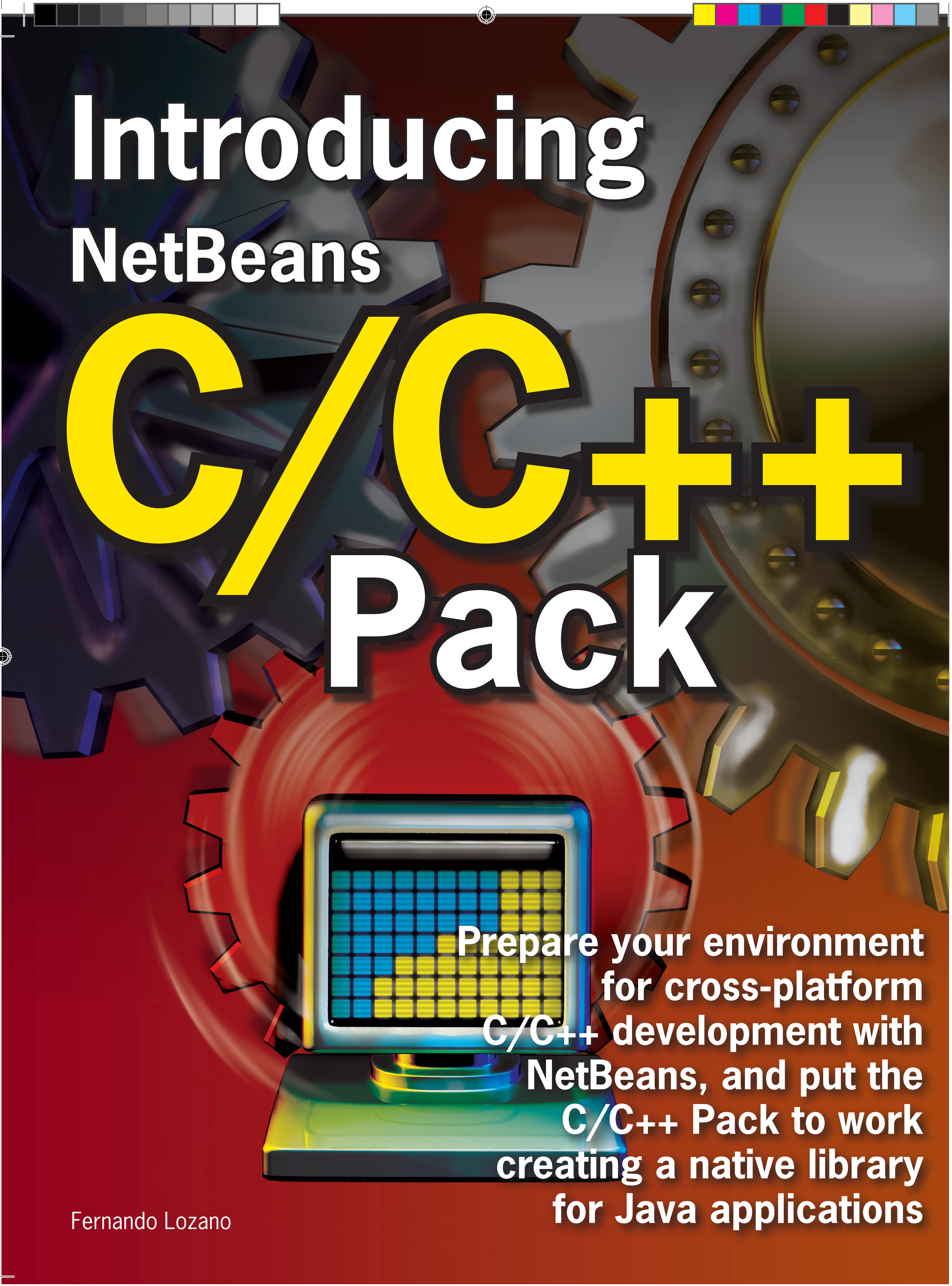
Conclusions

In the past, writing components to integrate with OpenOffice.org required an extensive setup of the NetBeans infrastructure, with steep learning curves. Everything was documented somewhere but putting this information together took far too much effort. This has changed with the new OpenOffice.org plugin module we've covered in this article. The module takes care of integration chores and lets developers concentrate on the implementation of their extensions. Also, the module's remote debugging capabilities make it much easier and faster to debug applications based on OpenOffice.org. If you need to integrate with OpenOffice.org or StarOffice, give it a try!

The next releases of the plugin will integrate Java more closely into the OpenOffice.org scripting framework, and will let you use Java for typical scripting related tasks, combining the advantages of an integrated scripting language with the power of NetBeans and Java technology.

Kay Koll (*kay.koll@sun.com*) is responsible for the technical marketing of StarOffice/ OpenOffice.org. He has been working in various positions for StarOffice since 1995. Kay lives in Hamburg, Germany.

Introducing NetBeans C/C++ Pack



Prepare your environment for cross-platform C/C++ development with NetBeans, and put the C/C++ Pack to work creating a native library for Java applications

Fernando Lozano

When NetBeans 5.5 was released in late 2006, it radically changed its own value proposition by offering first-class support for a language that doesn't run inside a JVM. The NetBeans C/C++ pack provided to C/C++ programmers most features Java developers were already used to: advanced source editing with syntax highlighting and code completion, built-in CVS support, hyperlinks to navigate function declarations, a class hierarchy browser, an integrated debugger, and integration with the *make* tool.

This article focuses on how the C/C++ pack can help Java developers. Although I'm sure you all would like to code the whole world in pure Java, reality frequently challenges us to interface with native code, be it legacy systems, a device vendor SDK or a high-performance math library. Also, sometimes we need to use native code to improve the user experience, by means of tighter integration with the underlying operating system. Wouldn't it be better to do all this from the same IDE we already use for Java development?

We'll show how to leverage NetBeans and the C/C++ Pack to develop portable native libraries using C/C++, and how to integrate them with Java code in a way that eases deployment in multiple platforms.

NetBeans C/C++ Pack is more than just C/C++ coding support for the Java developer. It also suits many native code projects very well. The **sidebar** "Other open source C/C++ IDEs" compares the Pack with some popular open-source IDEs for C/C++.

Installing NetBeans C/C++ Pack

Installing the C/C++ Pack per se will be a no-brainer for most users. No matter if you've installed the NetBeans IDE using the zip package or one of the native installers, you only need to run C/C++ Pack's installer and point it to your NetBeans IDE installation directory. (Note that, although the C/C++ Pack is mostly Java code with just one tiny native library, there's no multiplatform zip archive like the ones provided for the IDE.)

The installer itself will work the same for all supported platforms: Windows, Linux and Solaris. But configuring your environment for using C/C++ Pack may not be so easy. Just like the core NetBeans IDE needs a compatible JDK installation, the C/C++ Pack will require a C/C++ compiler and standard libraries and headers. So you need to install and configure these in advance.

To meet the Pack's prerequisites, we'll rely on the popular suite formed by the GNU C Compiler (GCC), GNU Binutils, GNU Make and GNU Debugger (GDB). This is the suite that received most of the QA effort of the C/C++ Pack developer team[1], and it's portable to Windows, Linux and Solaris environments.

Using the same compiler suite for all platforms greatly simplifies dealing with portable (and even non-portable) C/C++ code, as you won't need to spend time fighting compiler directives, runtime library inconsistencies and language dialects. Besides, you'll find that in most cases the GNU toolset competes head-to-head with other C compilers in both speed and optimization quality.

Installing the GNU toolset on Linux

Linux users should have no problem obtaining the GNU toolset for their preferred platform. Mine is Fedora Core 6, and as I installed a "development workstation" using Anaconda I already had everything ready for NetBeans C/C++ Pack. Users who didn't install Linux development tools when configuring their systems should have no problem using either *yum*, *up2date*, *yast* or *apt* to install the GNU toolset.

Stay clear of CD-bootable mini-distros like Knoppix for real development work. Instead, install a full-featured distro in a native Linux partition in your main hard disk. The few additional gigabytes used will prove to be a small cost for all the hassle you'll avoid.

Solaris users will also find it easy to install the GNU toolset; there are detailed instructions on the NetBeans Web site. But be warned:

netbeans.org/products/cplusplus

NetBeans C/C++ Pack home page

[1] The only other compiler suite supported so far is the Sun Studio product for Solaris and Linux.

if you think you'd be better served by the native platform C compiler (Sun Studio), think again. This is because NetBeans C/C++ Pack's debugger *needs* the GNU Debugger, and GDB has some issues running code generated by Sun compilers. So you can use Sun's compiler to produce final code, but you'd better use the GNU toolchain for development.

Installing the GNU toolset on Windows

Windows users won't be able to use native C/C++ compilers from Microsoft, Borland or Intel, and will have to stick with a Windows port of the GNU toolset. There are two options: Cygwin and MinGW.

The C/C++ Pack's docs at *netbeans.org* provide detailed instructions for using Cygwin, but I strongly advise you to use MinGW instead. The reason is that Cygwin relies on a Unix emulation layer, while MinGW uses native Windows DLLs for everything. Code compiled with Cygwin uses the standard GNU runtime library (*glibc*) on an emulation of Unix system calls, and semantics like mount points, pipes and path separators. But code compiled with MinGW will use standard Microsoft runtime libraries such as *MSVCRT.DLL*.

Cygwin has its uses, as many Linux and Unix software (specially open-source software) that has not yet been ported to Windows is easy to run under Cygwin without virtualization overhead. But I doubt you'd want to compromise stability and compatibility with the native platform when developing native libraries for use with Java applications. So MinGW is the way to go. The **sidebar** "Installing MinGW" provides detailed instructions.

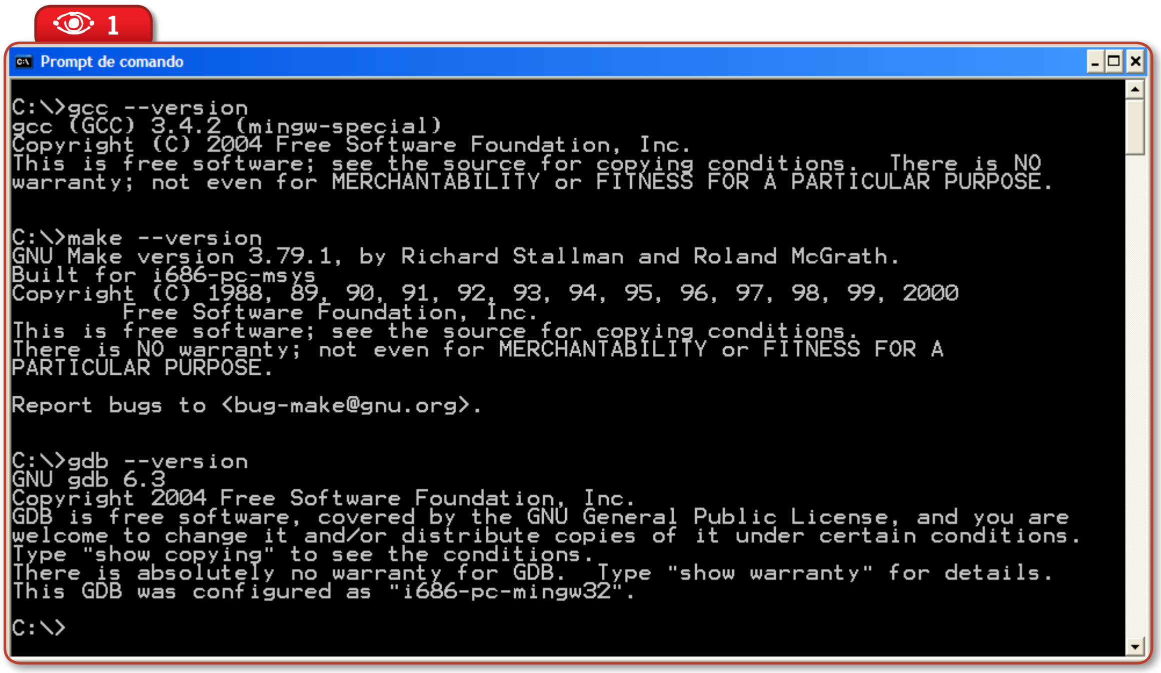
Checking prerequisites

Whatever your platform of choice, you need access to the GNU toolset from your operating system command prompt. It may be necessary to configure the system PATH before using NetBeans C/C++ Pack. You can check that you have all prerequisites are available before proceeding by using the commands displayed in **Figure 1**. (Although this figure shows a Windows command prompt, you'll be able to run the same commands from either the Linux or Solaris shells.) If you get software releases older than the ones shown, consider upgrading your GNU toolset.

When pure Java is not enough

Now that you have NetBeans C/C++ installed and its prerequisites configured, let's present this article's use case. You're

1



```
C:\>gcc --version
gcc (GCC) 3.4.2 (mingw-special)
Copyright (C) 2004 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


C:\>make --version
GNU Make version 3.79.1, by Richard Stallman and Roland McGrath.
Built for i686-pc-msys
Copyright (C) 1988, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 2000
        Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.

Report bugs to <bug-make@gnu.org>.


C:\>gdb --version
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-mingw32".

C:\>
```

Figure 1 Verifying that the GNU toolset is installed and configured correctly, and is using compatible releases.

developing a desktop Java application with cryptographic features, which saves sensitive data such as key rings and private keys in a local file system folder. You want to be sure that only the user who's running the application can read (and of course write) files to that folder.

The standard Java libraries provide methods in the **java.io.File** class for checking if a file can be read or written by the current user, but these methods don't check if other users can also read or write the same files. There are new methods in Java SE 6 that deal with file permissions, and work in progress under JSR 293; but if your application has to support Java 5 or 1.4, there's no escaping from native code. So our application will use native system calls to verify local folder permissions during initialization, and refuse to start if it finds the folder is not secure.

Java doesn't provide an easy way to declare external methods, like Free Pascal or Visual Basic, but it does of course provide the Java Native Interface, a standard and portable way to call native code from Java and vice versa. With the above use case in mind, we have to design an abstraction that hides platform details and the corresponding native code from the higher application layers. In the end, the apparent complexity of dealing with JNI may actually be an advantage, because it forces us to design the interface between Java and native code, instead of just going ahead and invoking operating system APIs directly.

The Java wrapper code

Let's get our feet wet. Start NetBeans, create a Java Class Library Project, and name it "OSlib". This project will contain all interfaces between our hypothetical application and the native operating system. Then create a new class named "FilePermissions", with the code shown in **Listing 1**.

The **native** keyword, you'll remember, means that the method's implementation will be provided by a native dynamic library. That library in our code is loaded by a static initializer in the class itself.

Following Test-Driven Development practices, I'll create unit tests instead of creating a test application for the OS interface. Right click *Test Packages* in the Projects window and select *New>Test for Existing Class* to generate a skeleton for testing the native method. Then change this skeleton to make it look like **Listing 2**.

The unit tests use a properties file (shown in the same listing) to get each test's target filesystem path. This way, all file paths can be easily changed to comply with native-platform naming conventions, without needing to recompile the tests themselves. Also, don't forget to create the target files and give them appropriate permissions.

If everything is fine so far, running the tests (by selecting the **FilePermissionsTest** class and pressing *Shift+F6*) should give the output shown in **Figure 2**. The **UnsatisfiedLinkError** exception is thrown because we haven't yet provided the native method implementation.

osix.net/modules/article/?id=670

A tutorial about how to develop native GUI Windows applications using MinGW

Other open-source C/C++ IDEs

C and C++ are of course much older than Java, and are still the languages of choice for many high-profile open-source projects. Based on that, on could guess there would be many other strong cross-platform and open-source C/C++ IDEs. You'll find that NetBeans C/C++ Pack may be the strongest one around, however. Let's look at some C/C++ Pack's competitors.

DevCPP

DevCPP is very popular among Windows developers. It's lightweight, well supported, and, like NetBeans, relies on external make tools and C/C++ compilers. Additionally, it supports a wide variety of C/C++ compilers. Though DevCPP is written using Borland Delphi, an attempt to port it to Linux (using Kylix) failed. So DevCPP is not an option for cross-platform C/C++ development.

OpenWatcom

The Watcom C/C++ compiler is cross-platform but offers no Unix support; it targets Windows and OS/2. Though not very user-friendly, it comes with an integrated debugger and a help system. It was once the compiler of choice for high-performance C/C++ applications, with its enhanced code optimizer and support for all Intel processor variants. When Sybase bought Watcom, though, the C/C++ compilers and IDEs fell into obscurity. Later the tools were released as open-source software. Nowadays, it looks like the community project is going well, but there's still no support for Unix and Linux systems. This makes OpenWatcom essentially a Windows-only IDE and not suitable for our purposes.

Anjuta

Anjuta is based on the complete GNU toolset for C/C++ development. In addition to the tools supported by C/C++ Pack, it supports the GNU Autotools, a set of scripts that simplifies generating Makefiles for multiple operating systems and compilers. It's also focused on GNOME development, so it provides templates for GTK, Gnome and Glade applications.

While DevCPP and OpenWatcom are Windows-only, Anjuta and KDeveloper (see next) are Unix-only. Some users have reported success running both under Cygwin, but they are still far from providing robust support for compiling and debugging native Windows applications.

For Unix developers, Anjuta provides integrated access to *man* pages and GNOME documentation. Its integrated debugger, like C/C++ Pack, relies on GDB. The latest releases provide integration with Glade, the Gnome visual UI builder.

KDevelop

Everything said before about Anjuta applies to KDevelop, if you just replace GTK/Glade/GNOME with Qt/QtDesigner/KDE. Anjuta and KDevelop are strong C/C++ IDEs for open-source desktops, but they don't cut it as cross-platform IDEs.

Eclipse CDT

C/C++ development support in Eclipse is almost as old as Eclipse IDE itself, but it has not matured as fast as the support for Java. Although currently labeled as release 4.0, Eclipse CDT doesn't provide many features beyond those in NetBeans C/C++ Pack (which is younger).

Also like NetBeans, Eclipse CDT doesn't integrate yet with visual development tools for Gnome, KDE or Windows. It has the advantage of supporting compilers other than the GNU compilers, but this won't be a real plus if your goal is developing cross-platform C code.

Red Hat is developing GNU Autotools and RPM generation plug-ins which, when they are released as production level, may become Eclipse CDT's real advantage over NetBeans C/C++ Pack (at least for Unix/Linux users). On the other hand, NetBeans is the development IDE for Open Solaris, so don't expect it to fall short in enhancements for Unix developers.

Conclusion

The only flaw one would find in C/C++ Pack, comparing it to other open-source alternatives for C/C++ development, is the lack of operating-system and third-party library documentation support in the help system. That would be also its main drawback when compared to proprietary C/C++ IDEs. But if you evaluate alternatives for cross-platform C/C++ development, the strongest (and only) competitor for NetBeans is also its main competitor in the Java space, that is, Eclipse.

jnimaker.dev.java.net

MKO JNI Stub Maker, a NetBeans plug-in for generating JNI headers

The native code project

Our unit tests are ready, but getting native code working alongside Java code is not trivial. We'll mock the native method implementation so we can focus on how to build a native library that can be called by Java code. Start by creating a C/C++ Dynamic Library Project, as shown in **Figure 3**. Name the project "NativeOSlib" and clear the "Set as main project" checkbox.

New C/C++ projects are created empty, except for a generated *Makefile* (see **Figure 4**), and are structured in virtual folders organized by file type – not by package names like Java projects. You'll be pleased to know that NetBeans C/C++ Pack includes a Makefile editor (even though there's still no support for running arbitrary Makefile targets as there is for Ant buildfiles).

Generating JNI Stubs

We're ready to begin writing our C code. First remember that all JNI-compliant native methods should use the declaration generated by JDK's *javah* tool. You could turn to the operating system command prompt to generate the C JNI stubs, but there's a better solution. It's the JNI Maker project, a plug-in module that adds a context menu for generating JNI header files from Java classes. Just get the *nbm* package from *jnimaker.dev.java.net* and install it using NetBeans's Update Center. After restarting the IDE, you should see a new menu item as shown in **Figure 5**.

Before generating JNI stubs, make sure you've built the Java project. JNI Maker uses the distribution JARs.

Now select *Generate JNI Stub* from the **FilePermissions** class's context menu. NetBeans shows a standard *File Save* dialog, where you can select a folder to save the generated *FilePermissions.h* header file. Move into the *NativeOSlib* project folder and create a new *src* folder (C/C++ Projects do not have a default file structure with separate source and test folders like Java projects do). Save the header file there. The output window will look like **Figure 6** if the operation is successful.

JNI Maker Release 1.0 will only work correctly under Windows, but the generated code will compile and run fine on Unix/Linux. The project developers have been contacted about the module's cross-platform issues and by the time you read this there should be a new release that will work on all platforms supported by NetBeans C/C++ Pack

Listing 1. FilePermissions.java – Utility class with a native method.

```
package org.netbeans.nbmag3.util;

import java.io.File;

public class FilePermissions {

  static {
    System.loadLibrary("NativeOSlib");
  }

  public FilePermissions() {}

  // Checks if a file or folder can only
  // be read/written by the current user
  public static native boolean isPrivate(String path);
}
```

Using the JNI Maker module has the same effect as running the following command from the operating system prompt, assuming the *OSlib* project folder is the current directory and *NativeOSlib* project folder is a sibling:

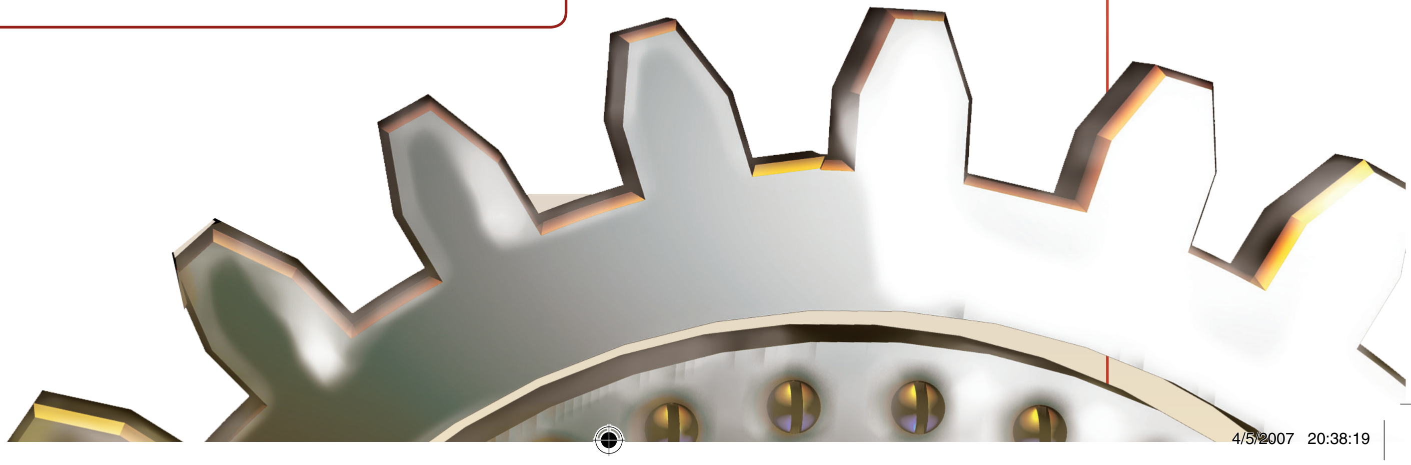
```
$ javah -classpath dist/OSlib.jar -jni -o ../NativeOSlib/src/
FilePermissions.h
    org.netbeans.nbmag3.util.FilePermissions
```

java.sun.com/javase/6/docs/technotes/guides/jni

Java Native Interface specification and related tips

mingw.sf.net

MinGW, the native GNU toolset for Windows



Listing 2. Unit tests for FilePermissions native methods

FilePermissionsTest.java

```
package org.netbeans.nbmag3.util;

import java.util.Properties;
import junit.framework.*;
import java.io.File;

public class FilePermissionsTest extends TestCase {
  Properties paths = null;

  public FilePermissionsTest(String testName) {
    super(testName);
  }

  protected void setUp() throws Exception {
    paths = new Properties();
    paths.load(this.getClass().getResourceAsStream(
       “/paths.properties”));
  }

  protected void tearDown() throws Exception {}

  public void testIsPrivateOk() {
    String fileName = paths.getProperty(
        “FilePermissions.test.privateOk”);

    assertTrue(“File does not exist”,
        new File(fileName).exists());

    boolean result = FilePermissions.isPrivate(fileName);
    assertEquals(true, result);
  }

  public void testCanReadButNotWrite() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            “FilePermissions.test.readButNotWrite”));
    assertEquals(false, result);
  }

  public void testCanBeReadByOthers() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            “FilePermissions.test.readByOthers”));
    assertEquals(false, result);
  }

  public void testCanBeWrittenByOthers() {
    boolean result = FilePermissions.isPrivate(
        paths.getProperty(
            “FilePermissions.test.writtenByOthers”));
    assertEquals(false, result);
  }
}
```

paths.properties

```
# For testing under Linux / Unix
FilePermissions.test.privateOk =
   /home/fernando/privateOk
FilePermissions.test.readButNotWrite =
   /home/fernando/readButNotWrite
FilePermissions.test.readByOthers =
   /home/fernando/readByOthers
FilePermissions.test.writtenByOthers =
   /home/fernando/writtenByOthers

# For testing under Windows
#FilePermissions.test.privateOk = C:\\test\\privateOk
#FilePermissions.test.readButNotWrite =
   C:\\test\\readButNotWrite
#FilePermissions.test.readByOthers =
   C:\\test\\readByOthers
#FilePermissions.test.writtenByOthers =
   C:\\test\\writtenByOthers
```

Installing MinGW

The MinGW project provides a native port of the GNU toolset for Windows platforms. Included in the base distribution are GNU C, C++, Objective-C, Ada, Java and Fortran compilers, plus an assembler and a linker; there's also support for dynamic libraries and Windows resource files. Additional packages provide useful tools like Red Hat Source Navigator, Insight GUI debugger and a handful of Unix ports like the *wget* download manager.

MinGW stands for “Minimalist GNU for Windows”. But it's “minimalist” only when compared to the Cygwin environment. (Cygwin tries to emulate a full Unix shell, complete with bash scripting, user commands and a Unix-like view of the filesystem.)

In fact, MinGW is complete to the point of providing Win32 API header files, and many popular open-source applications like Firefox have their Windows releases compiled using it. (Recent Cygwin releases include many MinGW enhancements as a cross-compiling feature, showing how Windows development is “alien” to MinGW alternatives.)

If you check the project's website, it looks like MinGW development has been stalled for quite a few years; the problem is that the site was automatically generated by a script that read the project's SourceForge area, and developers simply got tired of catching up with *sf.net's* design changes. However, MinGW is a very healthy project with active mailing lists and frequent file releases.

There is an installer for the base distribution named *mingw-x.x.exe* that downloads selected packages from SourceForge and installs them. The same installer can be used to update an existing MinGW installation.

Individual packages are downloaded to the same folder where the installer was

Other open-source C/C++ IDEs

bloodshed.net/devcpp.html, openwatcom.org, anjuta.sf.net, kdevelop.org

SourceForge file releases for MinGW

sf.net/project/showfiles.php?group_id=2435

Cygwin, a GNU toolset for Windows.

cygwin.com

started. This allows you to later copy the entire folder to another workstation and install MinGW there without the need of an Internet connection. Most extra packages provide their own installers or can simply be unpacked over an existing MinGW installation.

To satisfy C/C++ Pack's prerequisites, you'll need to download and install three MinGW packages: the base distribution itself, the GDB debugger, and the MSys distribution.

Installing MinGW

Download *MinGW-5.1.3.exe* (or newer) from the project's current file releases at *sf.net/project/showfiles.php?group_id=2435*, then launch it to see a standard Windows installer.

On the third step of the wizard (the second screen in **Figure S1**) you only need to select "MinGW base tools" and optionally "g++ compiler". Also, the Java Compiler may be interesting to play with, because of its ability to generate native machine code from Java sources and bytecodes, but it's not supported by NetBeans yet. Interestingly, the g77 (Fortran) compiler will be officially supported very soon.

After downloading all selected packages, the installer will ask for the destination directory and unpack all packages there. It's left to the user to configure environment variables so that MinGW tools can be used from the Windows command prompt.

Installing GDB

As we've seen, NetBeans C/C++ Pack needs GDB to be able to debug C programs. The MinGW distribution packages GDB as a stand-alone installer.

At the time of writing, the latest stable MinGW package for GDB was release 5.2.1, which won't refresh the NetBeans debugger's Local Variables window correctly. To solve this, download *gdb-6.3-2.exe* (or newer) from MinGW Snapshot Releases to a temporary folder and run it. Though you don't need to install GDB over MinGW, your life will be easier if you do, as you won't need to add another folder to your PATH system environment variable.

Installing MSys

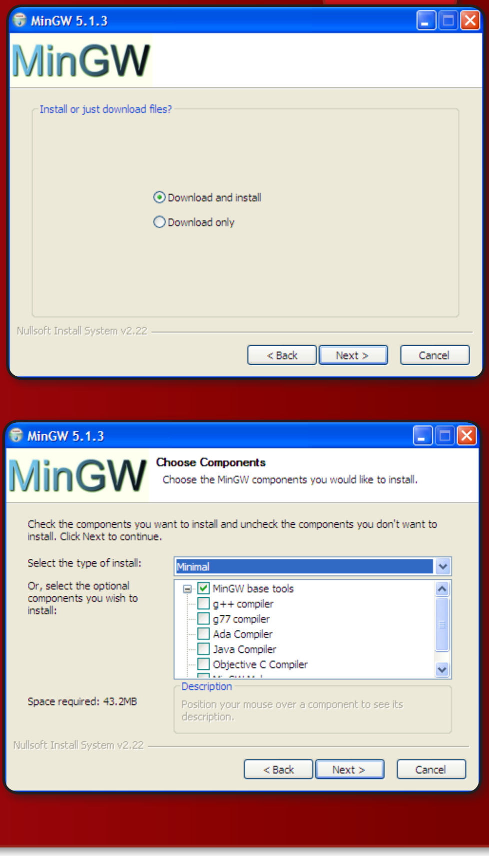
The MinGW base distribution already includes a make tool named *mingw32-make.exe*, but NetBeans C/C++ Pack won't be happy with it. MinGW's make tool is patched to be more compatible with other native Windows C compilers, and NetBeans expects a Unix-style make tool. NetBeans generated Makefiles even expect to find standard Unix file utilities such as *cp* and *rm*.

The MinGW MSys package satisfies these dependencies. It is a "Minimal System" that provides a Unix-style shell and file utilities, and allows open-source projects based on GNU Autotools to be easily built using MinGW.

Download *msys-1.0.10.exe* or newer to a temporary folder and start it. At the final installation step, a batch script configures the integration between MSys and MinGW. You will still have to add the MSys programs folder to the system PATH (in my case, *E:\MSys\1.0\bin*), as you did for the MinGW base distribution.

That's it. After running three installers and downloading about 23 MB, we are ready to develop C/C++ applications and libraries using the NetBeans IDE and C/C++ Pack on Windows.

S1



Install or just download files?

● Download and install

○ Download only

Choose Components

Choose the MinGW components you would like to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select the type of install: Minimal

Or, select the optional components you wish to install:

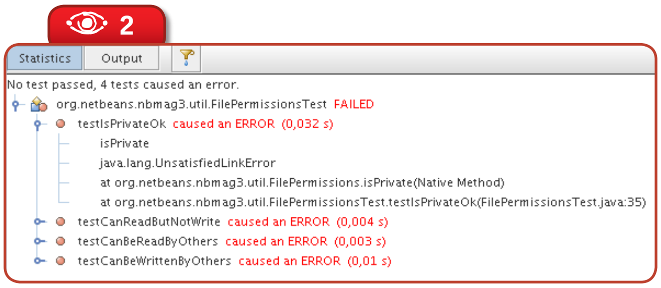
- ☑ MinGW base tools
- □ g++ compiler
- □ g77 compiler
- □ Ada Compiler
- □ Java Compiler
- □ Objective C Compiler

Space required: 43.2MB

Description

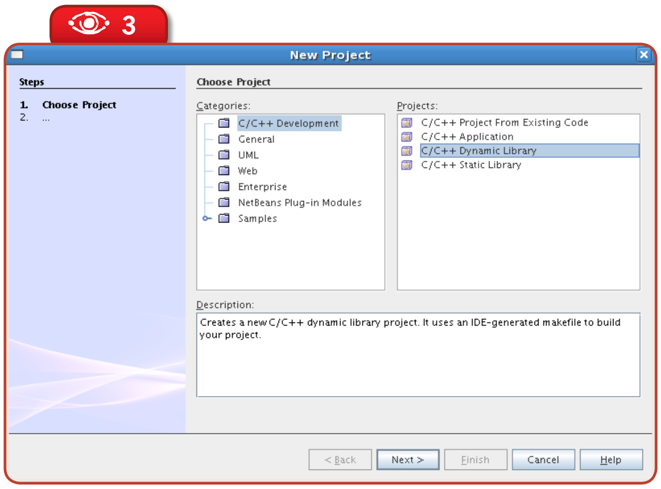
Position your mouse over a component to see its description.

Figure S1
Screens from MinGW's base distribution installer.



```
No test passed, 4 tests caused an error.
org.netbeans.nbmag3.util.FilePermissionsTest  FAILED
    testIsPrivateOk  caused an ERROR  (0,032 s)
        isPrivate
        java.lang.UnsatisfiedLinkError
        at org.netbeans.nbmag3.util.FilePermissions.isPrivate(Native Method)
        at org.netbeans.nbmag3.util.FilePermissionsTest.testIsPrivateOk(FilePermissionsTest.java:35)
    testCanReadButNotWrite  caused an ERROR  (0,004 s)
    testCanBeReadByOthers  caused an ERROR  (0,003 s)
    testCanBeWrittenByOthers  caused an ERROR  (0,01 s)
```

Figure 2
Running JUnit tests for the unfinished native method.



New Project

Steps
1. Choose Project
2. ...

Choose Project

Categories: C/C++ Development, General, UML, Web, Enterprise, NetBeans Plug-in Modules, Samples

Projects: C/C++ Project From Existing Code, C/C++ Application, C/C++ Dynamic Library, C/C++ Static Library

Description:
Creates a new C/C++ dynamic library project. It uses an IDE-generated makefile to build your project.

Figure 3
Creating a C/C++ Project in NetBeans.

(The command is broken to fit the column width, but it should be typed in a single line, of course.)

Now add the generated C header file to the NativeOSlib project. Right click *Header Files* inside the *NativeOSlib* project folder in NetBeans' Projects window, and select *Add Existing Item*. Then browse to the file *src/FilePermissions.h* and open it. The code will look like **Listing 3**.

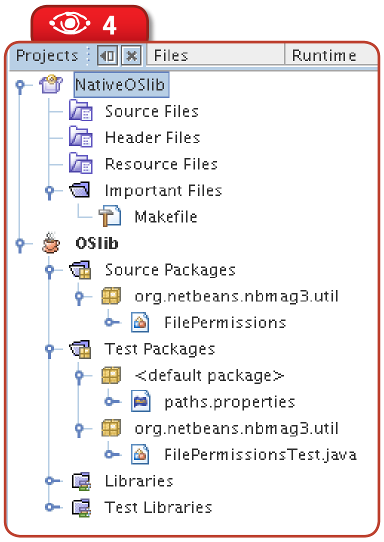
Mocking native code

Due to space constraints, we won't show you the final C code for the **FilePermissions.isPrivate()** native method, but the sources available for download will provide working implementations for both Windows and Unix (Posix) systems.

To create the C implementation file, right click *Source Files* and select *New>Empty C File*, then type "FilePermissions.c" as the file name and "src" as the folder name. A new node named *FilePermissions.c* should be created under *Source Files*.

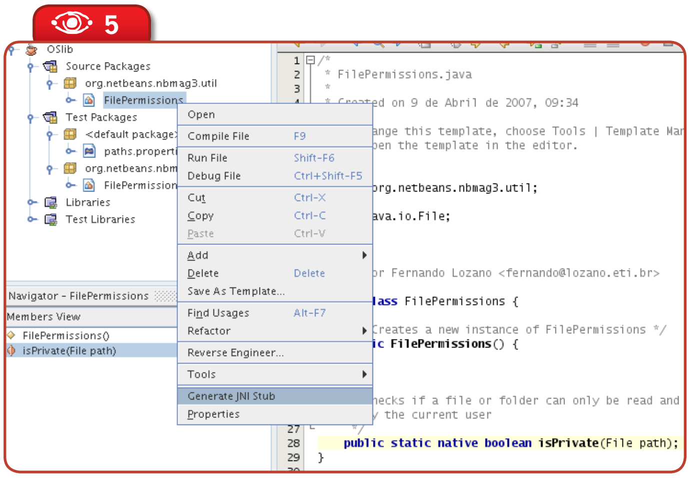
Copy the C stub function declaration from *FilePermissions.h* to *FilePermissions.c* and change it to include the header file. Also add parameter names. The code should look like **Listing 4**. (**Listing 3** highlights the declaration you have to copy, and **Listing 4** highlights the changes after copying.)

At this point, Unix and Linux users should be ready to build the native code and run unit tests again[2]. But Windows users first have to change a few project properties to make MinGW generate Windows-compatible JNI DLLs. The **sidebar** "JNI and MinGW" details these configurations.



```
NativeOSlib
    Source Files
    Header Files
    Resource Files
    Important Files
        Makefile
OSlib
    Source Packages
        org.netbeans.nbmag3.util
            FilePermissions
    Test Packages
        <default package>
            paths.properties
        org.netbeans.nbmag3.util
            FilePermissionsTest.java
    Libraries
    Test Libraries
```

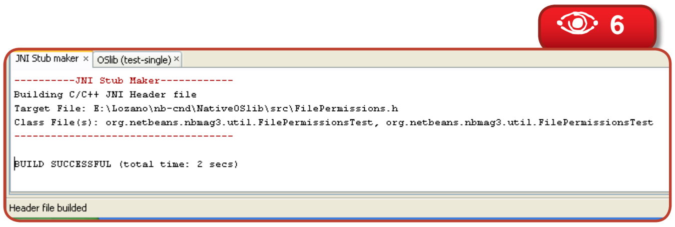
Figure 4
The new C/C++ project in NetBeans' Projects window.



```
OSlib
    Source Packages
        org.netbeans.nbmag3.util
            FilePermissions
    Test Packages
        <default package>
            paths.properti
        org.netbeans.nbm
            FilePermission
    Libraries
    Test Libraries
```

```
/*
 * FilePermissions.java
 *
 * Created on 9 de Abril de 2007, 09:34
ange this template, choose Tools | Template Mar
pen the template in the editor.
org.netbeans.nbmag3.util;
ava.io.File;
or Fernando Lozano <fernando@lozano.eti.br>
lass FilePermissions {
Creates a new instance of FilePermissions */
ic FilePermissions() {
hecks if a file or folder can only be read and
y the current user
     */
    public static native boolean isPrivate(File path);
}
```

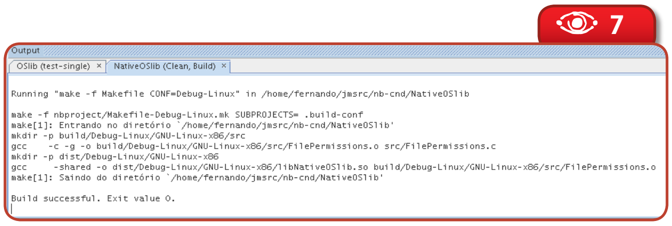
Figure 5
Generating a JNI stub using the JNI Maker plug-in module.



```
---------JNI Stub Maker-----------
Building C/C++ JNI Header file
Target File: E:\Lozano\nb-cnd\NativeOSlib\src\FilePermissions.h
Class File(s): org.netbeans.nbmag3.util.FilePermissionsTest, org.netbeans.nbmag3.util.FilePermissionsTest
----------------------------------

BUILD SUCCESSFUL (total time: 2 secs)
```

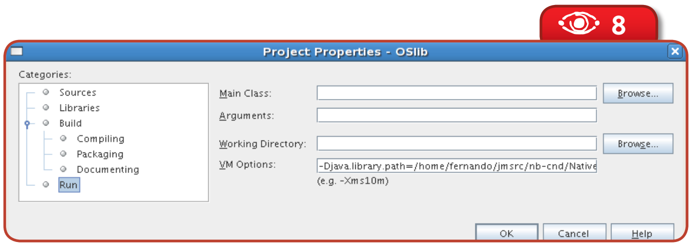
Header file builded

Figure 6 Output from the Generate JNI Stub command.



```
Running "make -f Makefile CONF=Debug-Linux" in /home/fernando/jmsrc/nb-cnd/NativeOSlib

make -f nbproject/Makefile-Debug-Linux.mk SUBPROJECTS= .build-conf
make[1]: Entrando no diretório `/home/fernando/jmsrc/nb-cnd/NativeOSlib'
mkdir -p build/Debug-Linux/GNU-Linux-x86/src
gcc    -c -g -o build/Debug-Linux/GNU-Linux-x86/src/FilePermissions.o src/FilePermissions.c
mkdir -p dist/Debug-Linux/GNU-Linux-x86
gcc     -shared -o dist/Debug-Linux/GNU-Linux-x86/libNativeOSlib.so build/Debug-Linux/GNU-Linux-x86/src/FilePermissions.o
make[1]: Saindo do diretório `/home/fernando/jmsrc/nb-cnd/NativeOSlib'

Build successful. Exit value 0.
```

Figure 7 Building the NativeOSlib project under Linux.



Main Class:
Arguments:
Working Directory:
VM Options: -Djava.library.path=/home/fernando/jmsrc/nb-cnd/Native
(e.g. -Xms10m)

Figure 8 Configuring the *java.library.path* property so unit tests can find the native code library on Linux.

Right click the *NativeOSlib* project and select *Clean and Build Project*. If there are no errors, you should see *make*'s output as in **Figure 7**.

Running unit tests again

You need to set the *OSlib* project's **java.library.path** system property before running it, or you'll still get **UnsatisfiedLinkError** exceptions. Open the project's Properties dialog, select the *Run* category and change *VM Options* to specify the full path to the *NativeOSlib* project's platform-specific native-library folder, which is inside the *dist* folder (see **Figure 8**). In Linux, this will be *PROJECT_HOME/dist/Debug/GNU-Linux-x86*; in Windows, *PROJECT_HOME\dist\Debug\GNU-Windows*.

Now run the unit tests again. The result should be as shown in **Figure 9**. Since the mock native code always returns true, some tests pass even if you have not created target test folders or forgotten to setup their access permissions. Anyway, the first test should fail because it takes an extra step to check if the target file path actually exists.

Managing platform-specific compiler settings

NetBeans C/C++ Pack puts object files in the *build* and *dist* folders, inside subdirectories named after the target platform, for example *GNU-Linux-x86* or *GNU-Windows*. But it won't save different compiler options for each target, forcing you to have a different project for each platform if there's a need for platform-specific compiler settings.

Listing 3. FilePermissions.h – JNI Stub for native methods in the FilePermissions class.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class
   org_netbeans_nbmag3_util_FilePermissions */

#ifndef _Included_org_netbeans_nbmag3_util_FilePermissions
#define _Included_org_netbeans_nbmag3_util_FilePermissions
#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class:     org_netbeans_nbmag3_util_FilePermissions
 * Method:    isPrivate
 * Signature: (Ljava/io/File;)Z
 */
JNIEXPORT jboolean JNICALL
Java_org_netbeans_nbmag3_util_FilePermissions_isPrivate(
   JNIEnv *, jclass, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 4. FilePermissions.h – JNI mock implementation for the FilePermissions native methods.

```
#include "FilePermissions.h"

JNIEXPORT jboolean JNICALL
Java_org_netbeans_nbmag3_util_FilePermissions_isPrivate(
   JNIEnv *env, jclass clazz, jstring path)
{
  return JNI_TRUE;
}
```

[2] At least if you use JDK packages compatible with your distro package manager, like the IBM and BEA JDKs provided by RHEL and SuSE Enterprise, or the RPM Packages from jpackage.org. If not, you'll have to add your JDK include folder to the GNU C compiler include directory. The configurations will be similar to the ones presented in the "JNI and MinGW" sidebar, but you won't need to change either the linker output file name or additional compiler options.

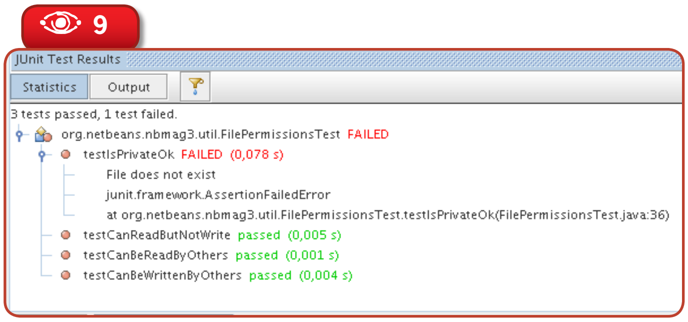


Figure 9 Running unit tests using a mock native implementation.

```
JUnit Test Results
Statistics | Output
3 tests passed, 1 test failed.
org.netbeans.nbmag3.util.FilePermissionsTest FAILED
  testIsPrivateOk FAILED (0,078 s)
      File does not exist
      junit.framework.AssertionFailedError
      at org.netbeans.nbmag3.util.FilePermissionsTest.testIsPrivateOk(FilePermissionsTest.java:36)
  testCanReadButNotWrite passed (0,005 s)
  testCanBeReadByOthers passed (0,001 s)
  testCanBeWrittenByOthers passed (0,004 s)
```

You can solve this using NetBeans C/C++ Pack's multiple configurations feature. Open *NativeOSlib*'s project properties and notice the *Configuration* combo box on the top of the window (**Figure 10**). The default configurations are meant to save different compiler settings for Debug and Release

JNI and MinGW

Unix and Windows native C/C++ compilers use different conventions for mangling function names[1]*, exporting global symbols from libraries and setting up stack frames. JNI on Windows uses Microsoft conventions for Windows DLLs, while GCC uses its own conventions for dynamic libraries. This means that if you simply try to compile and link a dynamic library, MinGW will stick to its Unix origins and produce a DLL that is incompatible with native Windows C/C++ compilers. The JVM won't be able to get native method implementations from that library and will generate more **UnsatisfiedLinkException**s.

The solution is to add a few command-line options when compiling C/C++ sources: **-D_JNI_IMPLEMENTATION_ -Wl,--kill-at**. Open the C/C++ Dynamic Library Project properties and expand *C/C++>Command Line*, then type these options in the *Additional Options* text field (see **Figure S1**).

You also need to add your JDK include folders (*JAVA_HOME\include* and *JAVA_HOME\include\win32*) to the project properties. Open *C/C++>GNU C Compiler>General* and change the *Include Directories* field as shown in **Figure S2**.

You need one last change in the C/C++ Dynamic Library Project properties so you can generate a JNI-compatible DLL. By default, NetBeans chooses a library name that corresponds to Cygwin conventions, but we need to use native Windows conventions. So you need to enter the *Linker>General* category and remove the "cyg" prefix from the *Output* field (**Figure S3**).

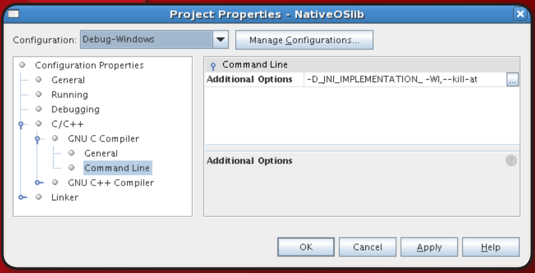


Figure S1 MinGW compiler options for generating JNI-compatible DLLs

Project Properties - NativeOSlib

Configuration: Debug-Windows

Command Line
Additional Options: -D_JNI_IMPLEMENTATION_ -Wl,--kill-at

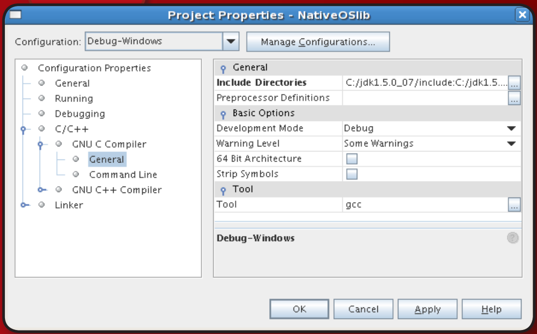


Figure S2 Configuring JDK include folders for MinGW

Project Properties - NativeOSlib

Configuration: Debug-Windows

General
Include Directories: C:/jdk1.5.0_07/include:C:/jdk1.5....
Preprocessor Definitions:
Basic Options
Development Mode: Debug
Warning Level: Some Warnings
64 Bit Architecture: □
Strip Symbols: □
Tool
Tool: gcc

Debug-Windows

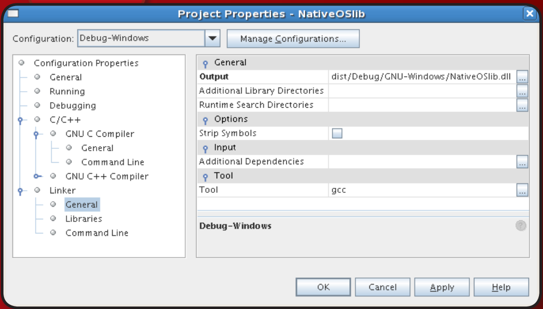


Figure S3 Changing the output file name for compliance with Windows DLL naming conventions

Project Properties - NativeOSlib

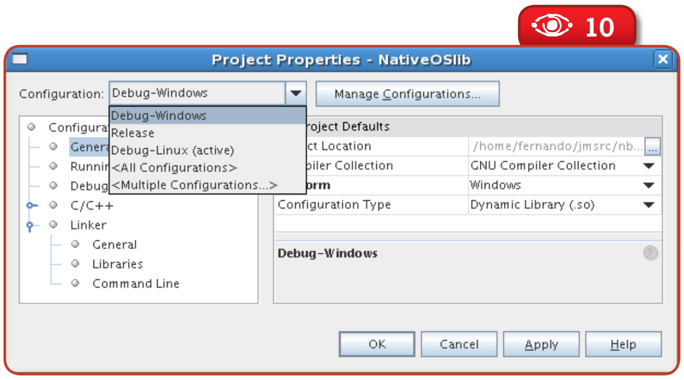
Configuration: Debug-Windows

General
Output: dist/Debug/GNU-Windows/NativeOSlib.dll
Additional Library Directories:
Runtime Search Directories:
Options
Strip Symbols: □
Input
Additional Dependencies:
Tool
Tool: gcc

Debug-Windows

[1] "Mangling" is the process used for generating public C++ function names in object files. It's needed because the C language doesn't support function overloading, and, to keep backward compatibility, C++ compilers generate a function name that encodes parameter types.

10



Project Properties - NativeOSlib

Configuration: Debug-Windows

Manage Configurations...

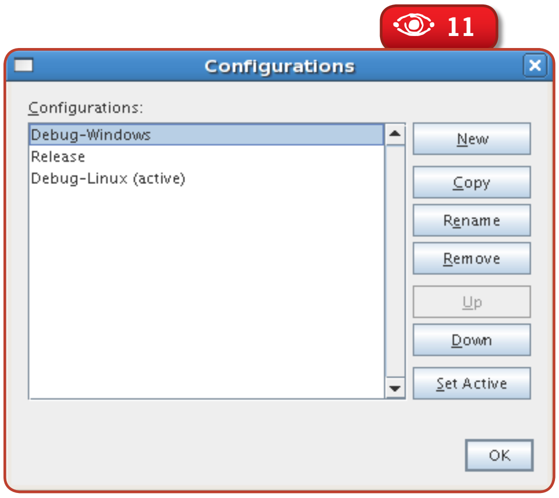
Debug-Windows
Release
Debug-Linux (active)
<All Configurations>
<Multiple Configurations...>

- Configura
  - Genera
  - Runnin
  - Debug
  - C/C++
  - Linker
    - General
    - Libraries
    - Command Line

roject Defaults	
ct Location	/home/fernando/jmsrc/nb...
piler Collection	GNU Compiler Collection
orm	Windows
Configuration Type	Dynamic Library (.so)

Debug-Windows

11



Configurations

Configurations:

Debug-Windows
Release
Debug-Linux (active)

builds, like keeping symbol information for Debug builds and optimizing code for Release builds. So if you want platform-specific configurations, you may need to create Release and Debug variants for each platform.

The *Manage Configurations* button to the side of the combo box lets you create new configurations either from scratch or as a copy of an existing configuration (see **Figure 11**). You'll notice I renamed the generated Debug configuration to *Debug-Linux* and copied it to a new configuration named *Debug-Windows*. Doing this lets you change the Windows configuration to include all options needed by MinGW for generating JNI-compatible DLLs, while keeping the default settings for the Linux configuration.

NetBeans-generated Makefiles provide many extension points (like the Ant buildfiles generated by the IDE), and they can be used outside the IDE. For example, for building the *Debug-Windows* configuration you'd type the following command at the operating system prompt:

```
make CONF=Debug-Windows
```

Thus, you could have Continuous Integration servers for many platforms, all being fed by the same CVS or Subversion source tree. And thanks to GNU C cross-compiler features it would be possible to have a "compile farm" that generates native binaries for multiple platforms, without the need for multiple OS installations. For example, a Linux server could generate both Windows and Solaris SPARC binaries.

Conclusions

NetBeans C/C++ Pack provides a rich environment for developing C and C++ applications and libraries. It's useful for Java developers that need to interface with native code and, of course, for developing fully-native applications. Compiler configuration may pose some challenges for Windows developers if they never tried GNU compilers before, but the effort will certainly pay off because of the increased portability of both code and Makefiles.

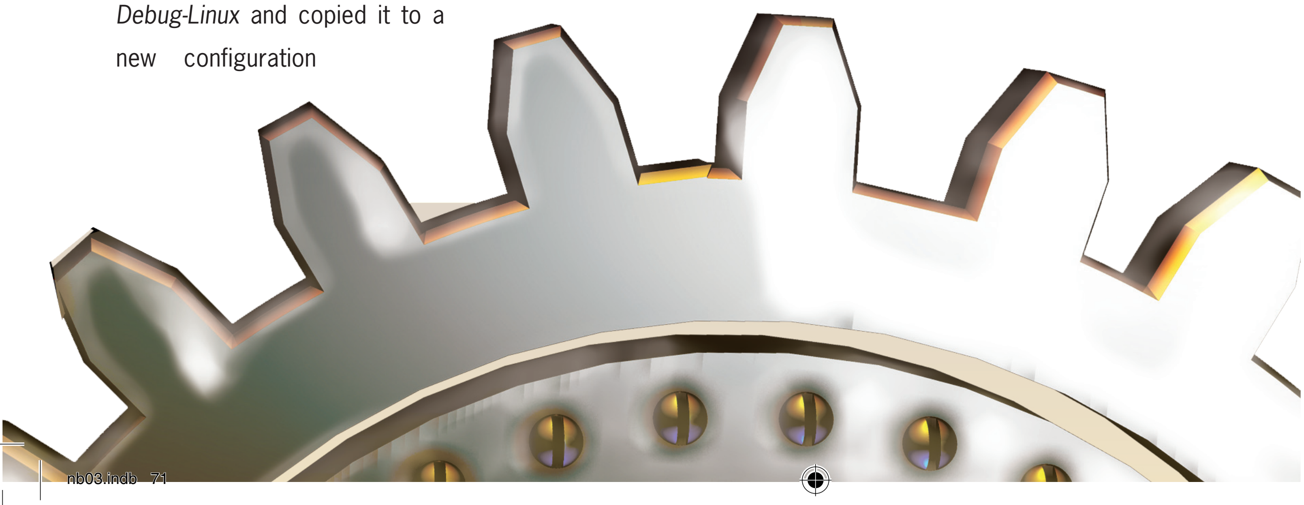
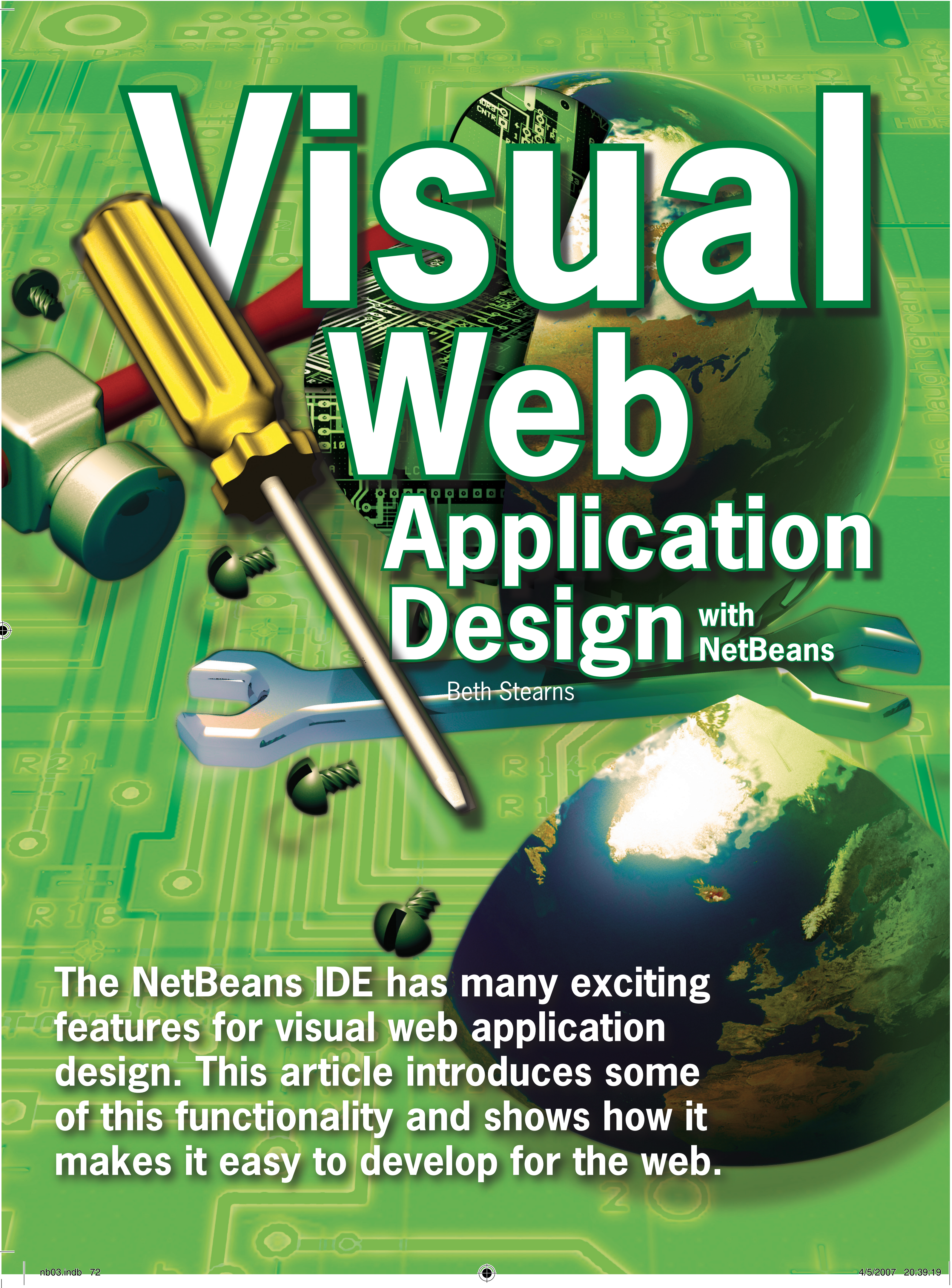


Figure 10
Combo box for changing compiler configurations for a C/C++ project.

Figure 11
Creating, renaming or copying configurations.

Fernando Lozano (*fernando@lozano.eti.br*) is an independent consultant and has worked with information systems since 1991. He's the Community Leader of the Linux Community at Java.net, webmaster for the Free Software Foundation and counselor to the Linux Professional Institute. Lozano helps many open-source projects and teaches at undergraduate and postgraduate college courses. He's also a technical writer and book author, as well as Contributing Editor at Java Magazine (Brazil) and freelance writer for other leading IT publications.



Visual Web Application Design with NetBeans

Beth Stearns

The NetBeans IDE has many exciting features for visual web application design. This article introduces some of this functionality and shows how it makes it easy to develop for the web.

The NetBeans IDE incorporates many features for visual web application design, a number of which have been available as part of the NetBeans IDE Visual Web Pack 5.5 module. NetBeans 6.0 integrates these visual design features directly into the IDE and adds more features to the mix.

This article gives you a quick overview of the feature highlights of the NetBeans visual web design environment, including the Visual Designer, Page Navigator, Query Editor, and Style Editor. We also show some of the things you can do easily and quickly with these tools when developing a web application.

Highlights

One of the best aspects of the NetBeans visual development environment is its flexibility. You can develop an application by first designing its individual pages, for which the IDE provides a Visual Designer with a palette of visual and non-visual components. You can add other elements to the palette, such as AJAX-enabled components you develop yourself or obtain from third parties, using the Component Library Manager. The Style Editor provides a graphical interface for perfecting the look of the different components.

You might prefer to first map out the application's logic flow instead of beginning with page design. If that's the case, you can start application design with the Page Navigator. Using the Page Navigator functions, you can create empty pages as stubs or placeholders and link them together in the Navigator window to define the application processing, and add the individual page layout and functionality later. You can even go back and forth between the Navigator and Visual Designer modes as you develop your application.

NetBeans also makes it simple for a web application to access a database. You can use the Query Editor functions to form complex SQL queries for applications that need to retrieve data from database tables or update a database. Other visual functions make it an easy matter to handle the display of data retrieved from database tables.

Although much is generated for you, there are still times you have to write your own custom code. When you are ready to write code, you can draw on the palette of generic code clips to help. You also have available all the shortcuts and other helpful features, such as code completion, that the Java source editor provides.

NetBeans 6.0 will introduce more visual design features, including the ability to develop portlets, add Enterprise JavaBeans components to applications, and incorporate web services. These features will rely on and extend the preexisting visual capabilities, so the learning curve for them should be minimal.

Using Components to Develop Web Application Pages

Page design is of course an important part of developing a web application. NetBeans provides a palette of visual design components that you drag and drop onto a page in the Design window, to set up your page quickly with the desired look and feel. In addition to these visual components, the IDE supports themes, which let you apply a predefined set of styles to visual components throughout a project, thus enabling you to change an application's entire appearance with a single mouse click.

As you design the underlying business logic, you can use the non-visual components – such as the converter, validator, and data provider components – to generate code. When you're ready to write the business logic, you can incorporate generic code snippets into your page bean by simply dropping code clips from the palette onto your bean code in the Java source code editor. All you need to do is add the correct variable names to these code clips.

You can easily create your own code clips too, just by selecting a snippet of code in the source editor and dragging it to the palette.

NetBeans displays the palette of components in a logical and in-

netbeans.org/kb/55/vwp-index.html#downloads/index.jsp

NetBeans 5.5 Visual Web Pack Documentation main page.

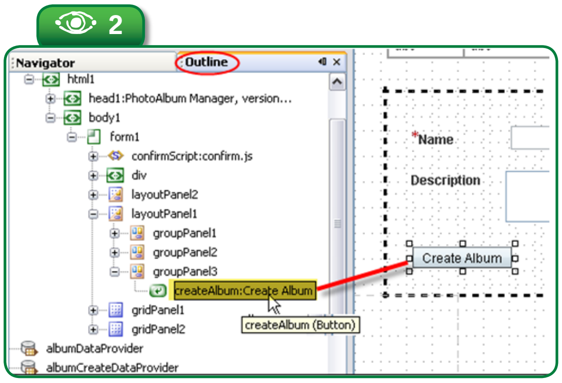


Figure 2 Viewing components in the Outline Pane

tuitive manner. The palette appears only when you're working on a web page for a project. When designing a web page – that is, when you have the page open in the Design window – the palette displays the components used to build the page, like the visual design components (buttons, drop-down lists, checkboxes, tables, etc.), as well as layout components, converters, validators, and data providers. If you used the Component Library Manager to add a set of components, you may have designated that they appear in their own separate category.

To illustrate, we use a sample media management project called PhotoAlbum, which is a web application through which users can manage different types of media (photos and audio data) stored in a local or remote database. You can use this application to organize media data into albums, such as photo albums. For example, if you want to organize a set of images, you can create multiple photo albums, upload these images into the different albums, and view the images as thumbnails or singly in a preview mode. (See the article "Developing a Media Management Application" at *netbeans.org/kb/55/photoalbum.html* for instructions on downloading this project zip file and installing and running the application.)

We opened the AlbumList page in the Design window and the palette displays the components we can use to build this page (see **Figure 1**). You add components to a page merely by dragging and dropping them on the page – or onto other components, such as in the case of validators and converters. The IDE ensures that no rules are broken, such as dropping a validator on the wrong type of component. Notice, too, that themes are not in the palette but are specific to individual projects and thus appear in the Projects pane.

At any time, you can go to the Outline pane to see the components added to a page. If you are unsure of a component's type, hover the mouse over it in the Outline pane to get more information. Or match its icon with the Palette icons (see **Figure 2**).

While you work on the page design and layout, you also can switch to editing the page's underlying Java code. To see the code in the Java editor, double click on the page background or select the Java view. When you change to the Java view, the Palette displays the available code clips (see **Figure 3**).

To work on a particular method, such as an action handler for a button, double click the button component in the Design window, and NetBeans displays the source code in the Java editor positioned at the button's action handler.

Designing pages

Let's examine how you might design a web application page using the visual components. The components in the Basic section of the Palette are typical GUI components: buttons, drop-down lists, checkboxes, hyperlinks, text areas, hyperlinks to images, and so forth. You use these components to add basic behaviors to your web pages.

These components provide properties to determine their appearance and behavior, and you can customize them using the component's Properties sheet or through dialogs. Although hand-editing a page's JSP code is possible (click

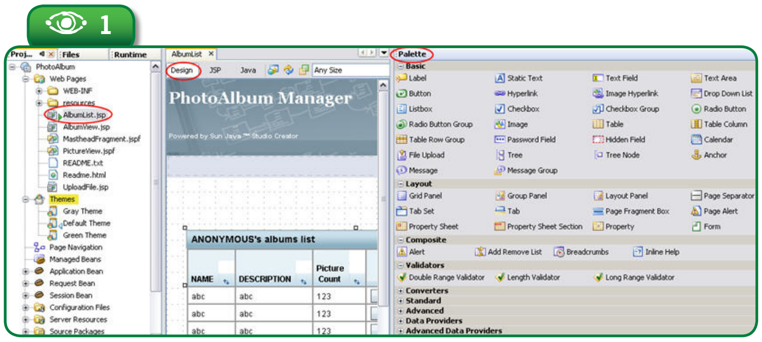
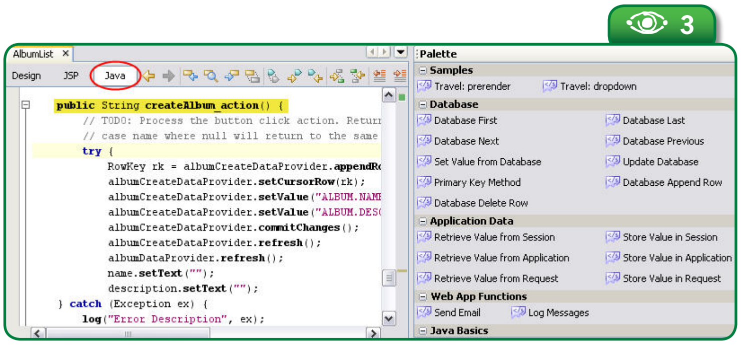


Figure 1 Palette of design components for a Web page



```
public String createAlbum_action() {
    // TODO: Process the button click action. Returr
    // case name where null will return to the same
    try {
        RowKey rk = albumCreateDataProvider.appendRo
        albumCreateDataProvider.setCursorRow(rk);
        albumCreateDataProvider.setValue("ALBUM.NAME
        albumCreateDataProvider.setValue("ALBUM.DES(
        albumCreateDataProvider.commitChanges();
        albumCreateDataProvider.refresh();
        albumDataProvider.refresh();
        name.setText("");
        description.setText("");
  } catch (Exception ex) {
    log("Error Description", ex);
```

Figure 3
Palette code clips

the JSP view to see and potentially modify the code), it is far easier and less error-prone to modify component properties in the Properties sheet.

The Layout components help you organize the appearance or layout of your page. Use them to add tab displays, alert boxes, grids, and forms – essentially to control the placement and alignment of other components on the page.

To give you an idea how all this works, let's take a closer look at some page-design issues frequently encountered by web application developers: setting up a consistent look across multiple pages of an application and aligning text and components on a page. Three of the layout components – Grid Panel, Layout Panel, and Page Fragment – handle this nicely.

Establishing a consistent look across pages

Use page fragments when you want to set up a consistent look to multiple pages of a web application. Page fragments are particularly useful for establishing uniform web page headers, footers, and sidebars. You design the page fragment once, then place it where appropriate on different web pages.

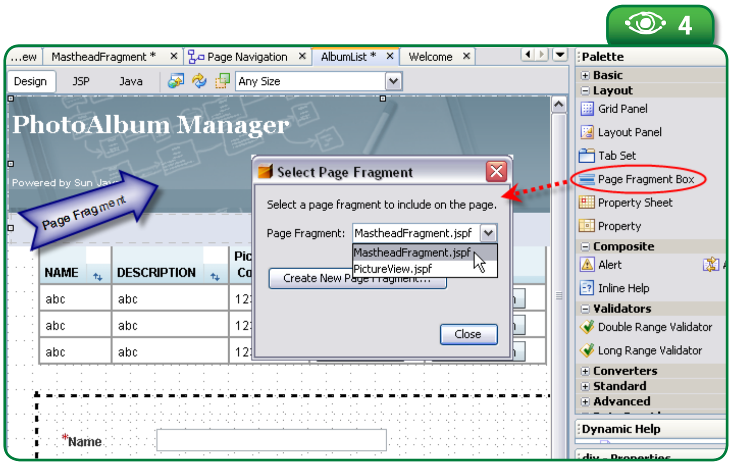
For example, you might want a consistent banner across all application pages. You define this look in a single page fragment, then include that fragment on the application pages. For a banner or masthead, you place the page fragment at the top of each web page. If you later make changes to it, these changes automatically appear on all the pages that include the page fragment.

Suppose you want to create a masthead banner for an application. Create a new page fragment: in the Projects pane, right click the project's Web Pages node and select *New>Page Fragment*. Then design your masthead by placing components in the fragment. Designing and building the page fragment is much the same as building a web page.

To add the masthead to other pages, drop a Page Fragment Box on a page, select the specific page fragment from the dialog, and position it where you want it to appear – at the top of the page in this case (see **Figure 4**). Fragments can also be copied and pasted among projects.

Controlling page layout

Grid Panel and Layout Panel components are useful for arranging text and other components. When you drop a Grid Panel on a page, it creates a table to which you then add other components. The added components display starting from left to right and from top to bottom. By default, a Grid Panel has one column and as many rows as needed to accommodate components dropped on it. You can change the number of columns and the display direction in the Grid Panel's Properties sheet.



Select a page fragment to include on the page.

Page Fragment: MastheadFragment.jspf

MastheadFragment.jspf
PictureView.jspf

Figure 4
Adding a page fragment to a Web page

A listing of reference articles and tips for Java Studio Creator IDE. Most of the information applies to the NetBeans 5.5 Visual Web Pack.

developers.sun.com/prodtech/javatools/jscreator/reference/index.jsp

Grid Panel components can also be nested within other Grid Panel components, giving you even finer control to position components on a page. To nest Grid Panels, drop a Grid Panel on top of a Grid Panel already placed on a page.

When dropping one component on top of another, be sure that the component already on the page is highlighted with a blue outline.

You can resize a Grid Panel directly on the page. For greater control, use the Style Editor, which you open by clicking the Grid Panel component's **style** property. You can use this property to set background color, margins, size, position, and so forth (see **Figure 5**). The Style Editor allows you to fine-tune the appearance of all the visual components.

The example MastheadFragment shown in **Figure 6** uses several Grid Panel components to control the display of its two hyperlinks (Home and Help) in the bottom panel. We used Grid Panels to place these links on the right, bottom side of the page. First, we dropped a Grid Panel (which we called "bottomPanel") on the page. Then we resized its width to match the top banner and its height to a size that would accommodate the images and text. Also, we set its **columns** property to two columns and set a background color.

Next, we dropped two Grid Panels on top of **bottomPanel**. Since **bottomPanel** displays two columns and the display direction is left-to-right, these two Grid Panel components display within **bottomPanel** from left to right. Also, the **bottomPanel** background color carries through as the background color for the components dropped onto it.

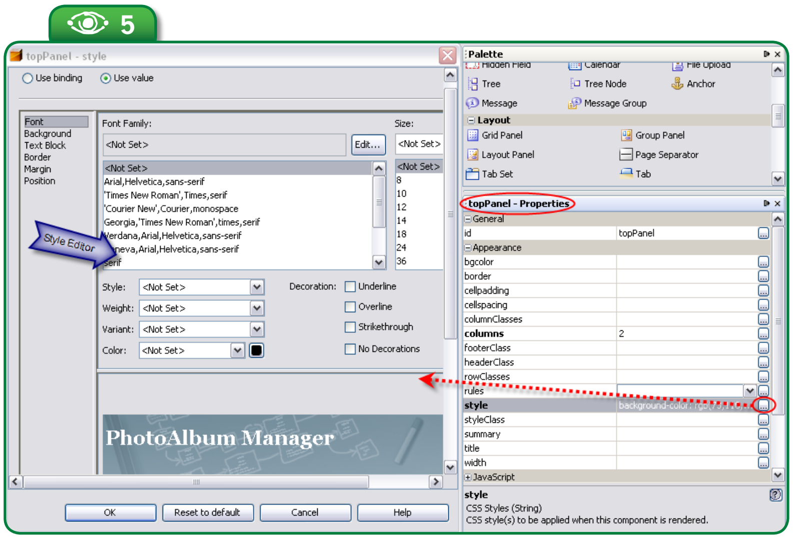
Of the two additional Grid Panels, the one in the left column (highlighted in yellow here) is a placeholder, and its width is set to cover the left half of the page. The Grid Panel on the right side holds the two links (see **Figure 6**). Since we want the links to display side-by-side, we changed the **columns** property to two for this Grid Panel.

After getting everything in position, we dropped two Image Hyperlink components onto the right panel. For each, we went to its Properties sheet and set its **text** property to the label – "Home" or "Help" – that we wanted to appear on the page. We also set each component's **imageURL** property to suitable image files for the display icons. In our application, these files are in the project's */resources* folder, but the **imageURL** property can point to wherever the files are located, of course.

We use the **textPosition** property to ensure that text is positioned to the right of the image associated with the hyperlink. (The **text**, **textPosition**, and **imageURL** properties are in the Properties sheet Appearance section.) We set the hyperlink's **url** property (found in the Behavior section) to the appropriate web page, so that a user is taken to that page when the link is clicked.

A Layout Panel component is much like a Grid Panel – you place a Lay-

5



topPanel - style

○ Use binding ● Use value

Font
Background
Text Block
Border
Margin
Position

Font Family: <Not Set> Edit...
Size: <Not Set>

<Not Set>
Arial,Helvetica,sans-serif
'Times New Roman',Times,serif
'Courier New',Courier,monospace
Georgia,'Times New Roman',times,serif
Verdana,Arial,Helvetica,sans-serif
Geneva,Arial,Helvetica,sans-serif
serif

<Not Set>
8
10
12
14
18
24
36

Style: <Not Set>
Weight: <Not Set>
Variant: <Not Set>
Color: <Not Set>

Decoration: □ Underline □ Overline □ Strikethrough □ No Decorations

topPanel - Properties

General
id topPanel
Appearance
bgcolor
border
cellpadding
cellspacing
columnClasses
columns 2
footerClass
headerClass
rowClasses
rules
style background-color: rgb(73,110,...
styleClass
summary
title
width
JavaScript

style
CSS Styles (String)
CSS style(s) to be applied when this component is rendered.

Figure 5 Style Editor with Grid Panel

6

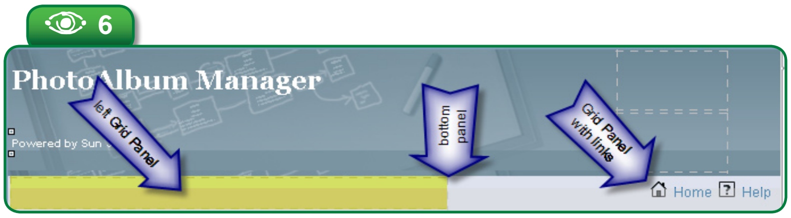
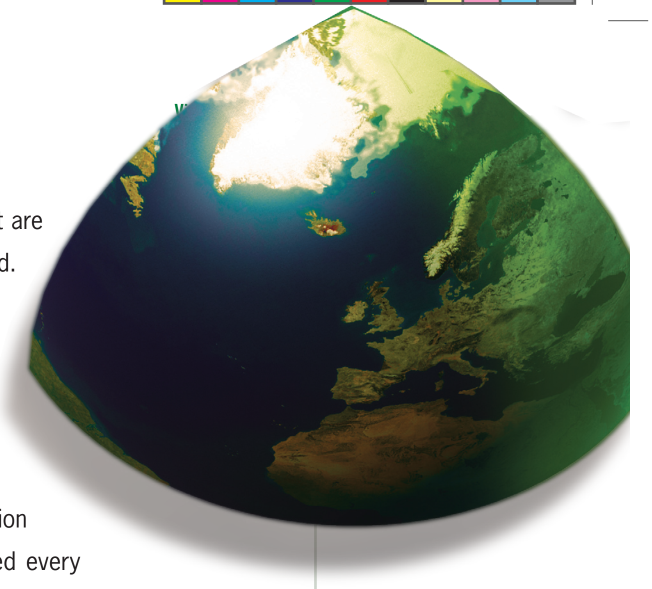


Figure 6 Controlling layout with multiple Grid Panels



out Panel on a page and then drop components onto the Layout Panel. But a Layout Panel gives you more flexibility in arranging components. Whereas a Group Panel can be placed on the same line as other components, a Layout Panel always appears on its own line, separated from components above and below it, when the page is rendered at runtime.

A Layout Panel lets you arrange added components in a flow or grid layout. When flow layout is used, the IDE places components dropped on a Layout Panel starting in the top left corner, adding components to the panel from left to right, across the top row of the panel until that row is filled. Subsequent components are added from left to right in the next row down, and so forth. You can drop a new component to the left of another component by hovering over the previously added component until a vertical mark appears to the left of that component.

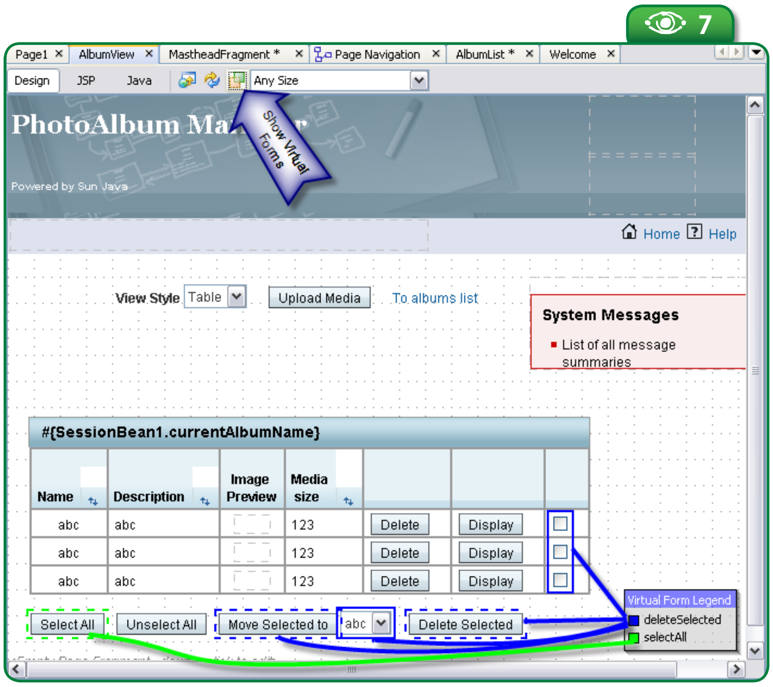
A Layout Panel works in grid layout mode only if the Snap to Grid option is set. You set this option from the *Tools>Options>Visual Designer* settings; at the same time, you can also customize the grid pattern size. When set to grid layout, added components appear in the panel aligned to the grid location at which they were added. Click the Align pop-up menu option for a Layout Panel (or for a component within a Layout Panel) to position components in the panel relative to the nearest grid corner.

Controlling user input

The Visual Designer includes a feature, called *virtual forms*, which lets you limit the portions of user input that are processed when a page is submitted. This is useful because you may have linked a number of input fields to validators (which means that the user-entered data in these fields is validated against some criteria when the page is submitted); but, given the application logic, you don't want all fields validated every time the page is submitted.

Let's look at how you might use virtual forms for a page. Our example page (see **Figure 7**) defines two virtual forms – **deleteSelected** and **selectAll**. Several components are included in these forms. Notice that the buttons, drop-down lists, and checkboxes for selecting, moving, and deleting files from an album are outlined in green or blue. Components outlined in green are part of the **selectAll** virtual form, while those outlined in blue are part of the **deleteSelected** virtual form.

A solid outline indicates an input component that participates in the virtual form, and a broken or dotted outline indicates a submission component, which is a component that, when clicked, submits the virtual form for processing. (To see the virtual forms legend for a page, toggle the virtual forms display icon at the top of the Design window, as shown in **Figure 7**)



Name	Description	Image Preview	Media size			
abc	abc		123	Delete	Display	□
abc	abc		123	Delete	Display	□
abc	abc		123	Delete	Display	□

7

Figure 7
Virtual forms displayed on a page

You add components to a virtual form via the dialog that opens when you click a component's pop-up menu *Configure Virtual Forms* option. The Configure Virtual Forms dialog shows that the Drop Down List component **albumList** participates in the **deleteSelected** virtual form. You can change whether a component participates or submits for any virtual form on the page by double clicking in the appropriate row and column. The Participate and Submit column entries display a Yes/No pull-down list if the selected component is of the right type, since only certain types of components can submit a page for processing. The New button in the dialog lets you create a new virtual form for the page (see **Figure 8**).

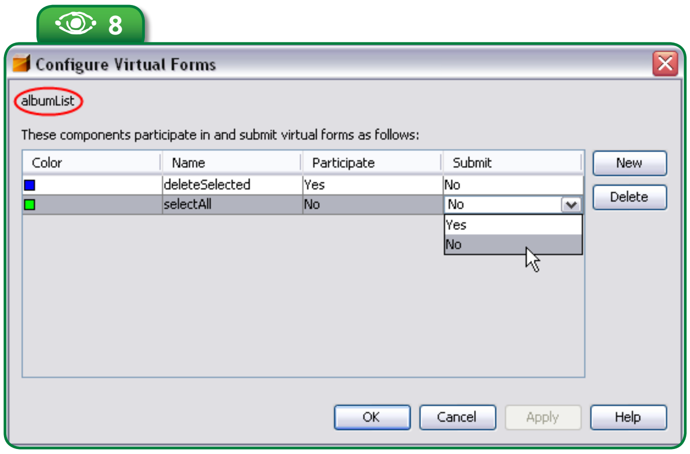
When the web page user interacts with a virtual form's submission component, such as by clicking the *Move Selected* to button, processing affects only the virtual form's input or participant components and ignores other input components on the page. In this manner, you can confine certain application operations to selected files or database table rows. For example, a user might check the boxes of several media files from the displayed list, then click the *Move Selected* to button with a target album designated from the drop-down list. The virtual forms feature ensures that subsequent processing moves only the selected files from the current album to the designated album.

Retrieving and displaying database data

The Visual Designer simplifies displaying tabular data on a page, and especially makes it easy to retrieve and display database data. To display tabular data, first drop a Table component from the Palette onto your page. The Visual Designer creates a generic three-column, multi-row table display (see **Figure 9**).

To have this component display data from a database table, you only need to drop the database table onto the generic table component on the page. (Database tables appear in the Runtime pane, beneath the Databases node. You must connect to the particular database or subschema before you can see its individual tables.)

Figure 8
Configuring virtual forms



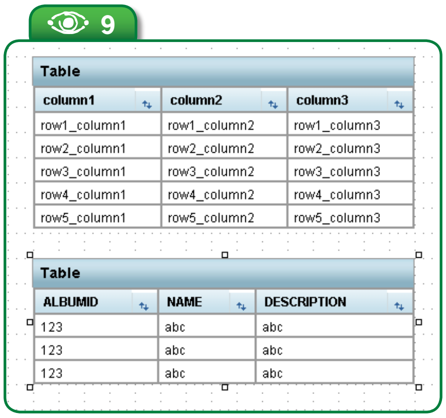
Configure Virtual Forms

albumList

These components participate in and submit virtual forms as follows:

Color	Name	Participate	Submit
🟦	deleteSelected	Yes	No
🟩	selectAll	No	No

Figure 9
Generic Table component and displaying a database table



Table

column1	column2	column3
row1_column1	row1_column2	row1_column3
row2_column1	row2_column2	row2_column3
row3_column1	row3_column2	row3_column3
row4_column1	row4_column2	row4_column3
row5_column1	row5_column2	row5_column3

Table

ALBUMID	NAME	DESCRIPTION
123	abc	abc
123	abc	abc
123	abc	abc

After you drop a database table on top of a generic table component, the Visual Designer binds the database table to the component and creates a default SQL query to retrieve the table data. The table display on the page changes to reflect the columns in the database table, while the rows indicate the type of data (again, see **Figure 9**).

When you bind a database table to a Table component, the IDE adds a **RowSet** to the project's Session Bean. (Every project has, in addition to its individual pages, a Session Bean and an Application Bean. The Session Bean maintains session scope variables, while the Application Bean is used for project-wide variables.) The **RowSet** includes a default SQL statement selecting all the columns in the database table. NetBeans also places the query's SQL statement in the Session Bean's constructor using the method **rowset.setCommand (String sqlCommand)**. Here, **rowset** is the **RowSet** for the table dropped on the page and **sqlCommand** is the query SELECT statement. Thus, you can edit the query in the Session Bean Java source code in the Java editor as well as through the Query Editor.

Customizing table data display

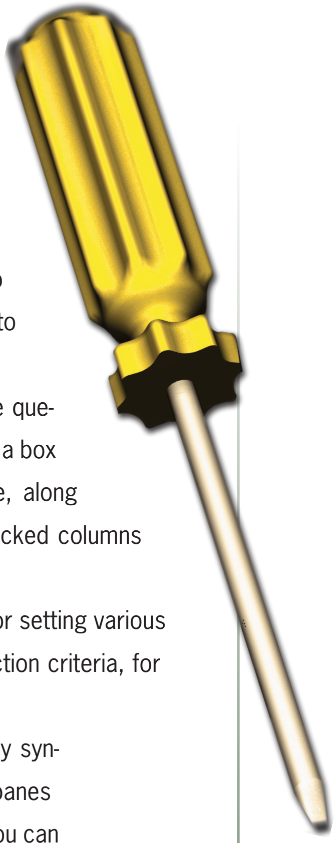
When you drop a database table on a Table component, the component display changes to show all columns from the database table in the same order they are returned from the database, and the database column names appear for the column headings. You can change the display using the Table Layout dialog, which you open from the Table's pop-up menu.

The Table Layout dialog consists of two tabs. These together give you options to remove columns, modify column headings, change column order, and even add new columns beyond what's in the database. From the Columns tab, use the Up/Down buttons to change the column display order, and use the left arrow to remove a selected column from the display (the double left arrow removes all columns). Use the right arrow to add available columns and New to create a new column. You can also modify column header text, column width, alignment, and sort capabilities (see **Figure 10**).

Setting the **sortable** option for a column enables users to sort the table data in real time by clicking the arrows in the column heading.

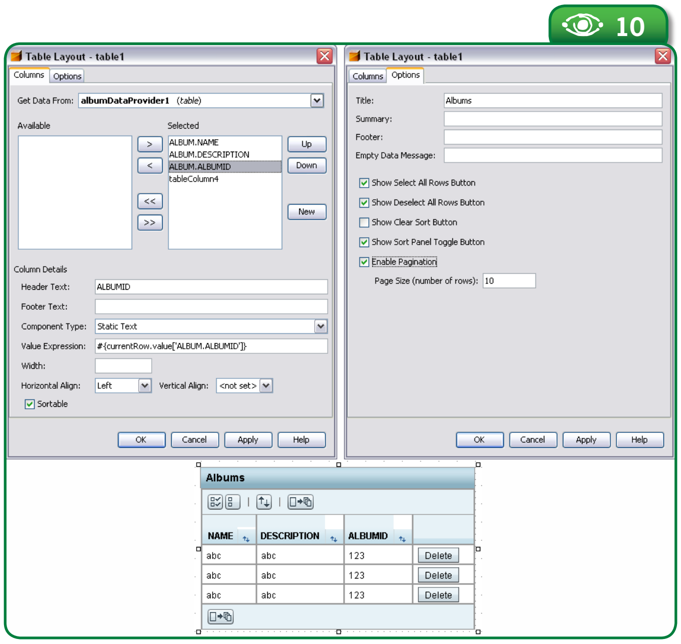
From the Table Layout Options tab, you can do such things as enable pagination, establish the page size (number of rows per page), and configure navigation buttons (select all rows, deselect all rows, clear sort, and so forth). These are automatically provided for you; no manual coding is required for the paging buttons.

Customizing SQL queries



The Query Editor provides a graphical interface through which you can edit and customize the SQL SELECT statement contained in a table's **RowSet** component. The Query Editor has four panes in which you can right click to bring up appropriate pop-up menus (see **Figure 11**). These are, from top to bottom:

- The **Diagram Pane** graphically represents the query. Each table dropped on the page appears as a box that indicates all the columns within the table, along with the table's primary and foreign keys. Checked columns are retrieved when you run the query.
- The **Grid Pane** displays a multicolumn table for setting various query conditions, such as sort order and selection criteria, for each table dropped on the page.
- The **SQL Pane** displays the actual SQL query syntax. Changes made in either of the top two panes are automatically reflected in all three panes. You can change the SQL query directly. The pane has a pop-up menu *Parse Query* option to update the other two panes. Use the *Run Query* option to test the query.
- The **Results Pane** at the bottom displays the results of a query you tested via *Run Query*.



Get Data From: **albumDataProvider1** (*table*)

Available

Selected: ALBUM.NAME, ALBUM.DESCRIPTION, ALBUM.ALBUMID, tableColumn4

Column Details

Header Text: ALBUMID

Footer Text:

Component Type: Static Text

Value Expression: #{currentRow.value['ALBUM.ALBUMID']}

Width:

Horizontal Align: Left Vertical Align: <not set>

☑ Sortable

Title: Albums

Summary:

Footer:

Empty Data Message:

☑ Show Select All Rows Button

☑ Show Deselect All Rows Button

□ Show Clear Sort Button

☑ Show Sort Panel Toggle Button

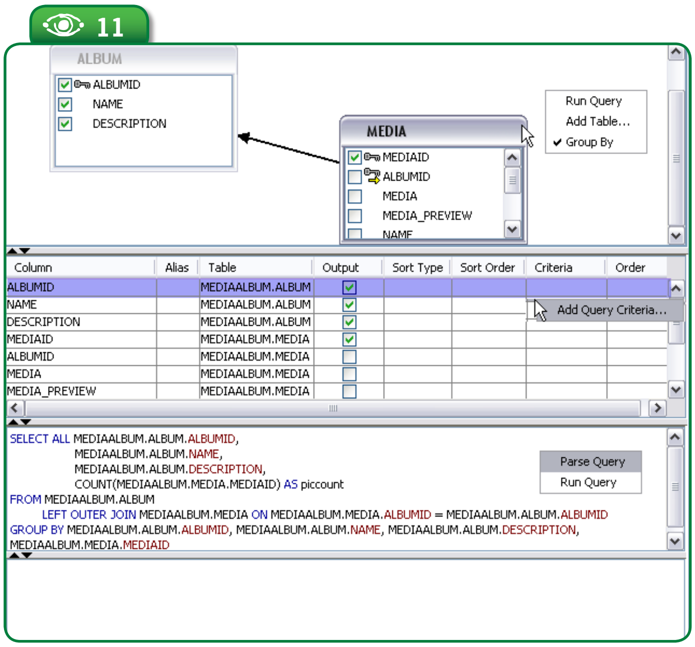
☑ Enable Pagination

Page Size (number of rows): 10

Albums

NAME	DESCRIPTION	ALBUMID	
abc	abc	123	Delete
abc	abc	123	Delete
abc	abc	123	Delete

Figure 10
Table Layout dialog



Column	Alias	Table	Output	Sort Type	Sort Order	Criteria	Order
ALBUMID		MEDIAALBUM.ALBUM	☑				
NAME		MEDIAALBUM.ALBUM	☑				
DESCRIPTION		MEDIAALBUM.ALBUM	☑				
MEDIAID		MEDIAALBUM.MEDIA	☑				
ALBUMID		MEDIAALBUM.MEDIA	□				
MEDIA		MEDIAALBUM.MEDIA	□				
MEDIA_PREVIEW		MEDIAALBUM.MEDIA	□				

```
SELECT ALL MEDIAALBUM.ALBUM.ALBUMID,
           MEDIAALBUM.ALBUM.NAME,
           MEDIAALBUM.ALBUM.DESCRIPTION,
           COUNT(MEDIAALBUM.MEDIA.MEDIAID) AS piccount
FROM MEDIAALBUM.ALBUM
     LEFT OUTER JOIN MEDIAALBUM.MEDIA ON MEDIAALBUM.MEDIA.ALBUMID = MEDIAALBUM.ALBUM.ALBUMID
GROUP BY MEDIAALBUM.ALBUM.ALBUMID, MEDIAALBUM.ALBUM.NAME, MEDIAALBUM.ALBUM.DESCRIPTION,
MEDIAALBUM.MEDIA.MEDIAID
```

Figure 11 Query Editor with sample SQL query

What are some of the query customizations you can do using the Query Editor? The Output column (in the grid pane) lets you select the columns to retrieve from the database and display. The Alias column lets you insert an AS clause into the SELECT clause. You can also control the sort order of the retrieved data.

The Diagram Pane's Add Table pop-up menu option lets you include additional tables in the query, essentially adding a JOIN clause to the SQL. The Group By option lets you group results by data type, which adds a GROUP BY clause to the query.

You can also limit a query to return only selected rows that meet some specified criteria. Use the *Add Query Criteria* pop-up menu option to open a dialog through which you add conditions for the query selection. You can select rows using comparison operators (such as equal to, greater than, less than, not equal to, and so forth), in addition to LIKE and IN. You can apply selection criteria to multiple table columns and specify the order to evaluate these criteria. Also, when you test a query for which you have specified selection criteria, a dialog prompts you to enter data for the selection parameters.

Defining application flow

Use Page Navigation to define the process flow of an application. Page Navigation, accessed from the Projects pane, lets you visually design the application flow by drawing links between the application pages and thus model the interactions of the application.

You can use the Page Navigation feature at any point in the application development process, and you can revisit Page Navigation to fine-tune interactions among pages, particularly when you want a specific action component on one page to link to another page.

Typically, you start to define the application flow by working at the page level, linking one page to another. Then, as you implement application details on the various pages – that is, place buttons, links, and so forth – you can return to the Navigation window and create navigational links at the component level within individual pages. You can even add buttons and other links to pages directly in the Navigation window.

The Navigator indicates the linkage between the pages of an application. (We illustrate the linkage between pages using the PhotoAlbum application, which includes several pages: *AlbumView.jsp*, *AlbumList.jsp*, *UploadFile.jsp*, among others. As with any application that has multiple pages, there needs to be a defined navigation between the pages.)

Click an individual page, such as we did for *AlbumList.jsp*, and the Navigator expands the page icon to show the individual linkable components on the page and where they link to, if anywhere (see **Figure 12**). Click and drag from one page to another to create a link between the two pages. Similarly, drag from a page's individual components, such as buttons, to another page to create links from the specific component on the first page to the second page. The Navigator adds the backing code to the pages and component action handler methods.

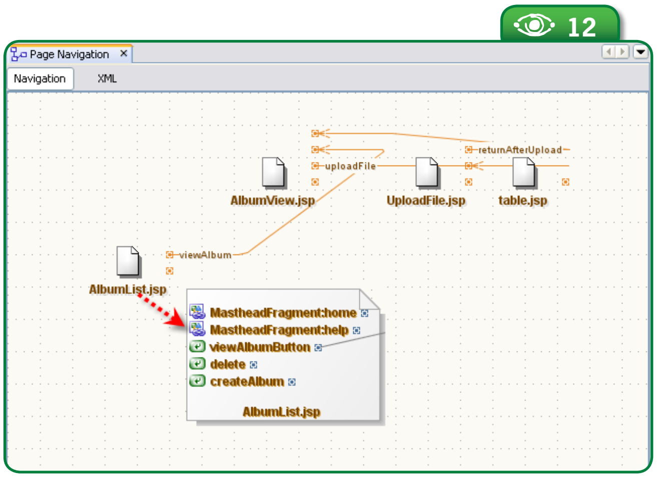


Figure 12
Page Navigation

Working with JavaScript code

Many web application developers want to work closely with JavaScript code. The Visual Designer makes it easy to invoke JavaScript from within an application. As you know, JavaScript code can be embedded within the JavaServer Pages code, where it is identified by its own set of tags.

When you work with JavaScript code, you bind the code to a component event so that the code is invoked when the event occurs. You can do this binding through the JavaScript section of a component's properties. Select the component and locate the event in the JavaScript section of its property sheet; then enter the JavaScript code directly, or click the "..." button to open an editor window.

To illustrate, suppose you want the user to confirm an action connected to a button click. First select the button component; then, in the JavaScript section of the component's Properties sheet, open the editor window of the target event, which in this case is **onClick**. You might enter the following JavaScript code:

```
return confirm (“Are you sure?”)
```

When you're finished, the component's Properties would look like **Figure 13**.

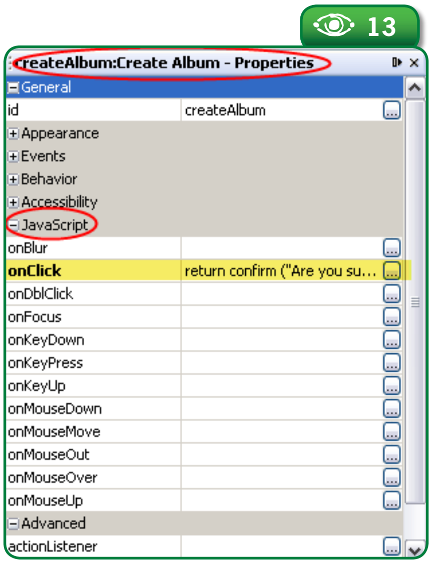
One good use of JavaScript code is to obtain confirmation from a user for an operation. For example, an application might bind JavaScript code to a delete button, to compel the user to respond to a confirmation message before the deletion occurs. JavaScript can also be used for actions that change component state, for validation checks, and for setting file names when retrieving data. The JavaScript code can be fairly complex, such as code for a button's validation checks (see **Figure 14**). Such complex code is best added using the Properties editor window.

The JavaScript you enter through the editor window is inserted into the JSP code for the component. For example, the JavaScript code for the button's **onClick** property shown in **Figure 14** appears as follows in the JSP:

```
<ui:button action=”#{AlbumView.btnMoveSelected_action}”
  binding=”#{AlbumView.btnMoveSelected}”
  id=”btnMoveSelected”
  onClick=”var i;&#xa;for (i=0;i&lt;document.forms[0].elements.length;i++)
  {&#xa;  if (document.forms[0].elements[i].checked){&#xa;
  return true;  &#xa;  }&#xa;}&#xa;alert(&quot;
     No items selected&quot;);&#xa;return false;”
     text=”Move Selected to”/>
```

When you use the Properties editor window to enter JavaScript code, the Visual Designer handles inserting the JavaScript into the JSP page with the proper tags and formatting. Of course, you can always edit the JSP yourself and manually add JavaScript code.

There are other ways to use JavaScript in an application. You can put



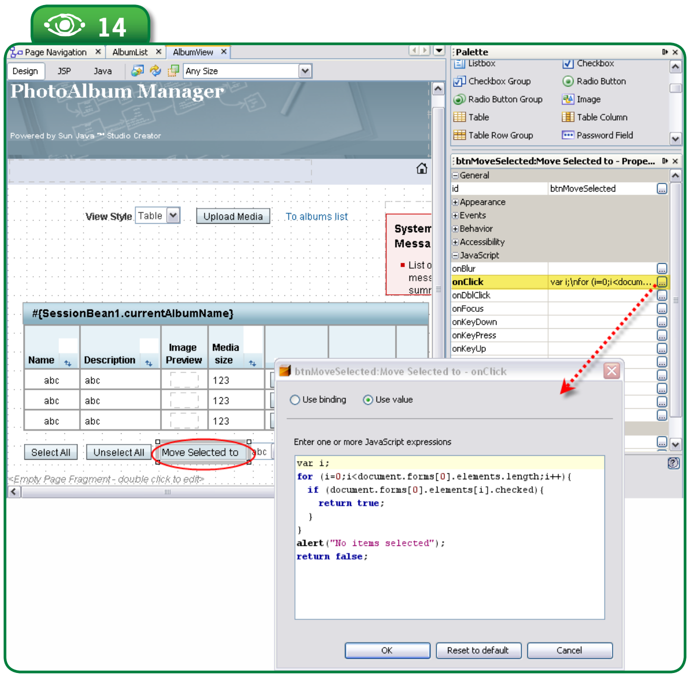
createAlbum:Create Album - Properties

General	
id	createAlbum
Appearance	
Events	
Behavior	
Accessibility	
JavaScript	
onBlur	
onClick	return confirm ("Are you su…
onDblClick	
onFocus	
onKeyDown	
onKeyPress	
onKeyUp	
onMouseDown	
onMouseMove	
onMouseOut	
onMouseOver	
onMouseUp	
Advanced	
actionListener	

Figure 13
Editing JavaScript properties

developers.sun.com/prodtech/javatools/jscreator/reference/pdf/creator_prog_guide.pdf

PDF compilation of the Java Studio Creator reference articles and tips.



id: btnMoveSelected

onClick: var i;\nfor (i=0;i<docum...

btnMoveSelected:Move Selected to - onClick

○ Use binding ● Use value

Enter one or more JavaScript expressions

```
var i;
for (i=0;i<document.forms[0].elements.length;i++){
  if (document.forms[0].elements[i].checked){
    return true;
  }
}
alert("No items selected");
return false;
```

Figure 14 JavaScript for button validation check

Beth Stearns (*beth.stearns@sun.com*) has been writing developer articles and books concerning the Java language since 1995. She has written extensively on Java Enterprise, JavaBeans, and Java Native Interface technologies as well as the Java Studio Creator and NetBeans development tools. Beth has co-authored numerous books in the Java Series, most recently, *Designing Web Services with the J2EE 1.4 Platform* and *Applying Enterprise JavaBeans, Second Edition.*

the code in a separate file and then use the Script component to bind to that file. First, enter the code to a file, giving it a *.js* extension, and then add the file to the project. (Place the file in a subdirectory of the project directory.) Next, drop a Script component, found in the Advanced section of the Palette, onto a page, and bind the Script component's **url** property to the *.js* file.

You can also drop the Script component onto the page and add your JavaScript code directly in the JSP editor. When added to a page, a Script component does not display but you should see the following line added to the JSP page:

```
<ui:script binding="#{AlbumView.script1}" id="script1"/>
```

You can then insert your JavaScript code within that tag, being sure to include the ending **</ui:script>** tag. For the simple confirmation shown before, for example, you might have:

```
<ui:script binding="#{AlbumView.script1}" id="script1"
  return confirm("Are you sure?")
</ui:script>
```

Although it is more complicated to use the Script component, it allows you to add generic JavaScript code that is not bound to an event.

Conclusions

This introductory article should have given you a good idea of how easy it is to develop visual web applications with NetBeans and its Visual Designer tools. The Visual Designer provides a palette of customizable components that you use to design web pages, along with a Style Editor to help you fine-tune their appearance. In addition, non-visual components provide user-input validation and conversion code automatically, plus a virtual forms feature that simplifies user input processing. There is also a palette of sample code clips that you can use when you have to write backing code. And if you are comfortable with JavaScript code, the Visual Designer facilitates writing JavaScript for different components.

You also have available to you a set of components for specifically working with database tables and tabular data, and these components make it a straightforward matter to retrieve and display database data. The Query Editor provides a graphical interface to help you develop more elaborate SQL queries.

Page Navigation is another feature that helps you design the flow your application. You use this interface to link pages together, either at the page level or at the component level.

All in all, these tools take care of many of the page layout and coding chores required to develop a web application. You can concentrate on getting the right look for your application's web pages and its business logic, and leave the rest to NetBeans.

NetBeans

NetBeans
magazine