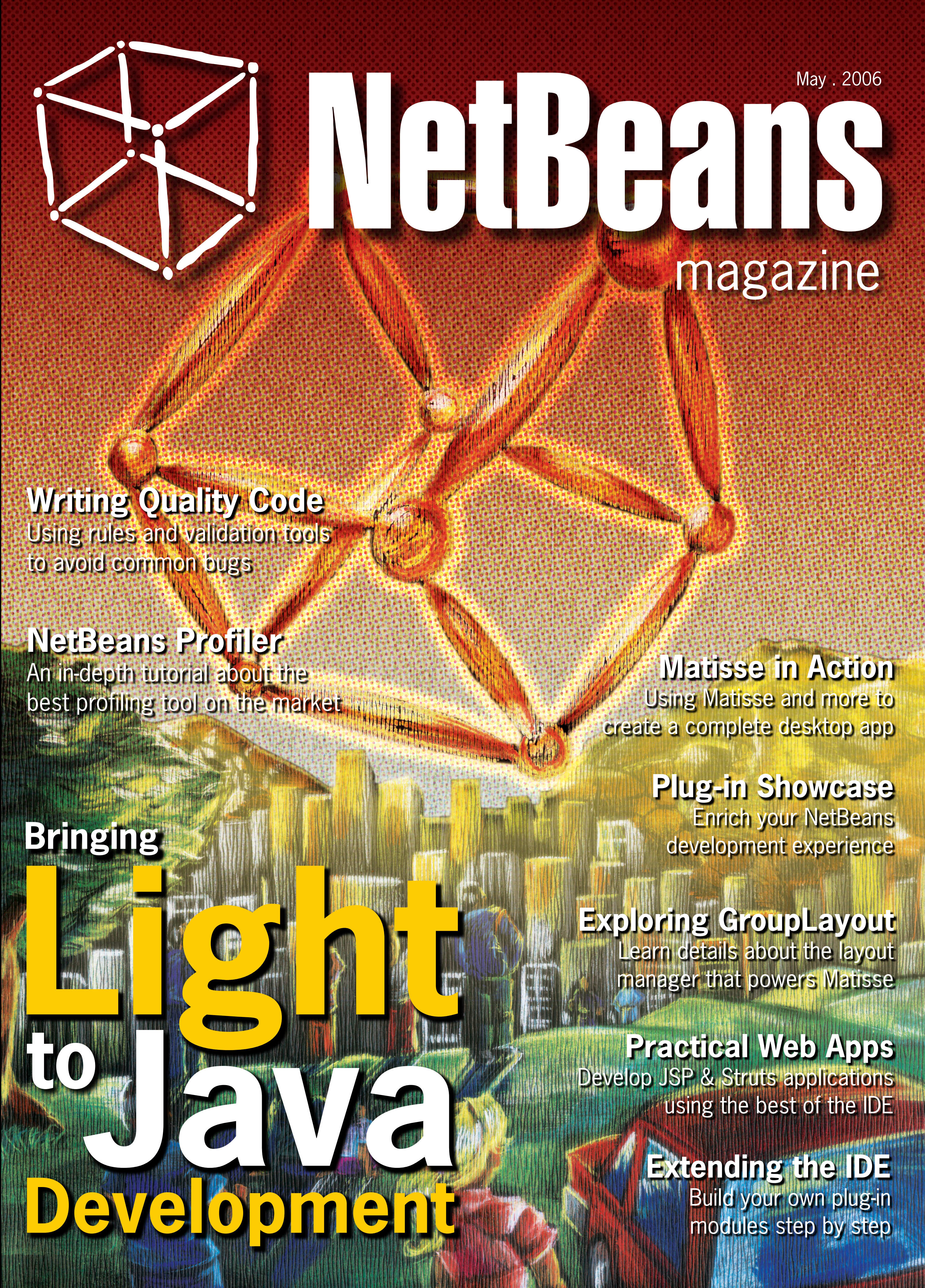
IDE Overview . Plug-in modules . Profiling . Matisse . GroupLayout . Web development

May . 2006

NetBeans magazine



Writing Quality Code
Using rules and validation tools to avoid common bugs

NetBeans Profiler
An in-depth tutorial about the best profiling tool on the market

Matisse in Action
Using Matisse and more to create a complete desktop app

Plug-in Showcase
Enrich your NetBeans development experience

Exploring GroupLayout
Learn details about the layout manager that powers Matisse

Practical Web Apps
Develop JSP & Struts applications using the best of the IDE

Extending the IDE
Build your own plug-in modules step by step

Bringing Light to Java Development

NetBeans magazine

Publisher & Editor-in-Chief
Leonardo Galvão
leonardo@javamagazine.com.br

Assistant Editor
Osvaldo Pinali Doederlein
opinali@gmail.com

Design and Layout
phDesign (*phdesign.com.br*)

Graphic Designers
Tarcísio Bannwart, Jaime Peters Jr, Tersis Zonato, Lais Pancote

Illustrators
Felipe Machado & Francisco Peixoto

Contributors
Fernando Lozano
Geertjan Wielenga
Gregg Sporar
Leonardo Galvão
Osvaldo Doederlein
Tomas Pavek

Editorial Support
Robert Demmer
John Jullion-Ceccarelli

NetBeans Magazine is supported by NetBeans.org

Produced in Brazil

Number One

The NetBeans project has been going through an unprecedented number of changes, broadening its scope, increasing quality and usability, and expanding communities and user adoption. In many areas, like Swing building or JME development, NetBeans IDE is now the tool to beat, with levels of functionality and productivity that match or exceed any other tool, open source or commercial.

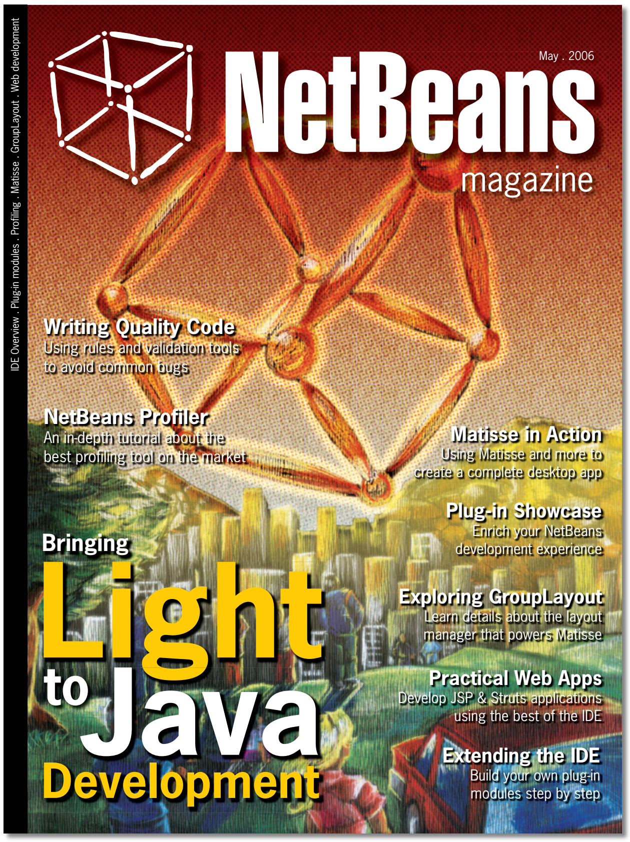
This special first edition of NetBeans Magazine showcases a wide selection of IDE and extension features, from desktop and web development to plug-in module creation. Beginners will learn how to develop a complete desktop application using Matisse and other IDE facilities. Seasoned programmers will also benefit, knowing details about the NetBeans Profiler, which introduces breakthrough innovations in profiling tools, and further learn about GroupLayout, the layout manager that is the core of Matisse. Also shown is how to use IDE features and modules to detect bug patterns, enforce code conventions, and closely follow coding rules that promote overall quality and reduce maintenance costs.

NetBeans IDE has always followed the “it just works” principle, aggregating all the functionality developers need from day to day. But there’s always some little niche necessity that has to be taken care of. The extensibility features of NetBeans come to the rescue, and the recent versions of the IDE make creating plug-in modules a breeze. Catering for the growing community of plug-in module fans, the magazine includes a special section describing tens of little and great extensions, which enable NetBeans developers to program in other languages, use new APIs and frameworks, and squeeze more functionality out of standard IDE features. And if you just can’t stand being in the user role for long, a tutorial shows how to create a new plug-in module from scratch.

NetBeans has gone very far and very fast – but still the community manages to increase the rhythm, with version 5.5 at the door and the first releases of 6.0 already available. The best part is you don’t get only to watch. You can join in, and participate in this movement that’s bringing light to Java development.

Happy coding,
Leonardo Galvão

Contents


IDE Overview . Plug-in modules . Profiling . Matisse . GroupLayout . Web development

May . 2006

NetBeans magazine

Writing Quality Code
Using rules and validation tools to avoid common bugs

NetBeans Profiler
An in-depth tutorial about the best profiling tool on the market

Matisse in Action
Using Matisse and more to create a complete desktop app

Plug-in Showcase
Enrich your NetBeans development experience

Bringing Light to Java Development

Exploring GroupLayout
Learn details about the layout manager that powers Matisse

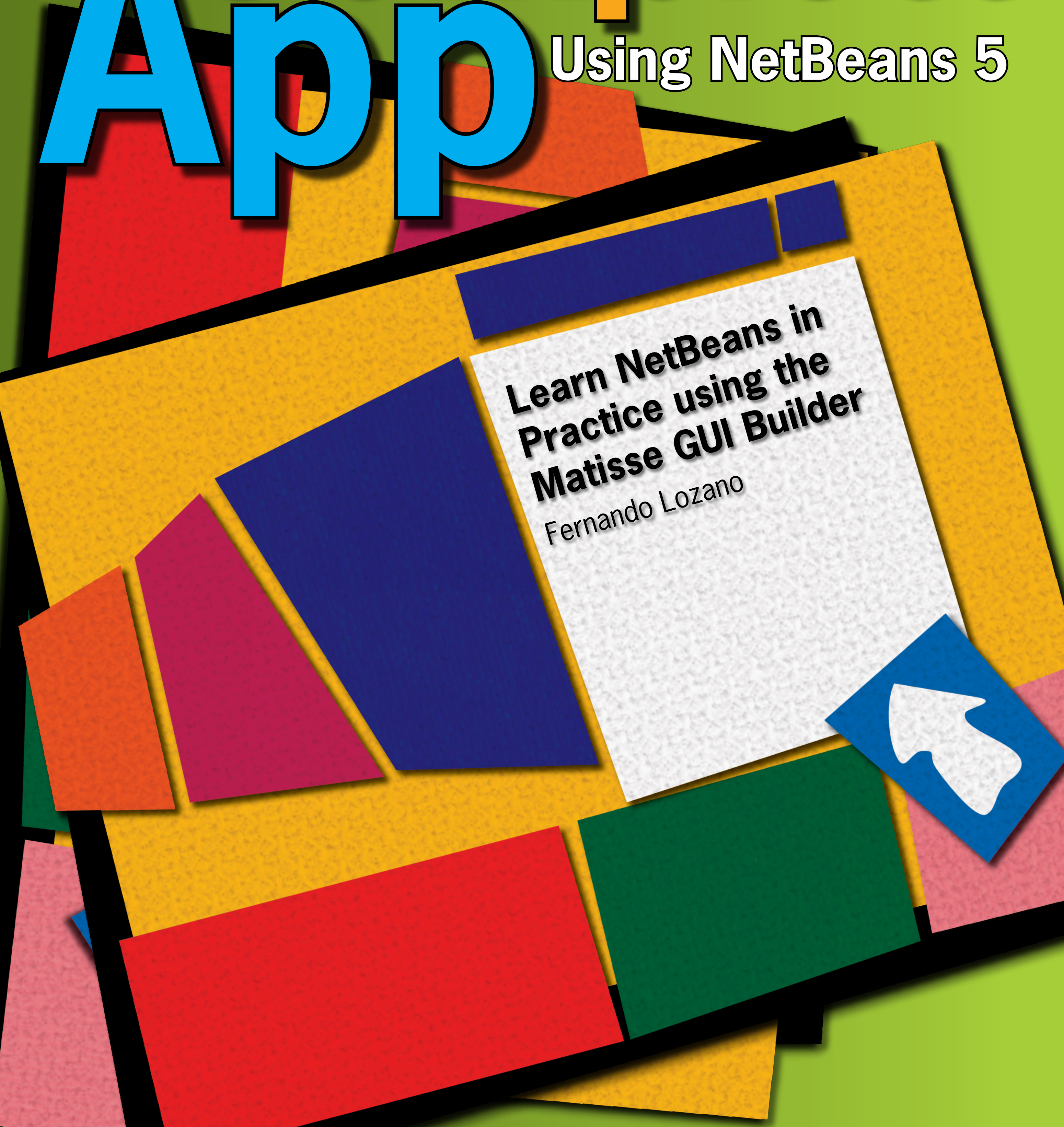
Practical Web Apps
Develop JSP & Struts applications using the best of the IDE

Extending the IDE
Build your own plug-in modules step by step

A Complete App Using NetBeans 5

Learn NetBeans in Practice using the Matisse GUI Builder

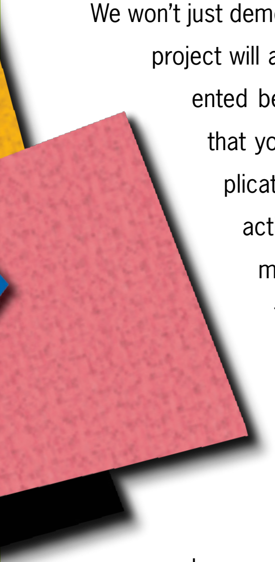
Fernando Lozano



NetBeans is not a newcomer to the Java arena. In fact, it is one of the oldest Java IDEs still available on the market. But the most exciting developments happened in the latest releases, specially 4.0 and 5.0, with the renewed commitment from Sun and participation of an ever-growing community of users and developers. In many respects, such as desktop development, NetBeans can be regarded as the most powerful and most easy-to-use Java IDE

This article gives an overview of the IDE while building a complete desktop application. Instead of a hello-world kind of app, we build a more "real-world" application: a to-do list commonly found as part of PIM suites. The application will use an embedded relational database and require customization of Swing components, so it will be a small-scale real project except only for the lack of help content and an installer.

We won't just demo the IDE features. The project will also stick to Object-Oriented best practices, showing that you can develop GUI applications quickly and interactively, without compromising long-term maintenance and a sound architecture. However, to keep the tutorial short we'll skip some practices usually required by corporate environments and well supported by NetBeans, such as test-driven development using JUnit tests, and source-control systems like CVS.



The reader will need basic Swing and JDBC skills, beyond familiarity with the Java language and Object-Oriented programming. We start with the basic procedures to install and configure NetBeans, including a quick tour of the IDE user interface. Then the sample application is presented, followed by the steps to create it using the IDE features.

The first part of this article will be more detailed, because the visual design capabilities are among NetBeans' strongest features. As we move deeper into the application logic, the article will switch to a higher level discussion. That way, this article aims for two objectives:

1. Provide newbie developers with an introduction to using the NetBeans IDE;

2. Provide more seasoned developers with useful insights about GUI development best practices, while using the best of NetBeans features.

The to-do application will be developed using a three-step process. The first step prototypes the UI design, where NetBeans really shines. The second step focuses on user interaction and event handling; it's actually a second prototype for the application. The third and last step builds the persistence and validation logic. Readers familiar with the MVC architecture will note these steps form a process that starts with the View, then builds the Controller, and finally builds the Model.

Installing NetBeans

Installing NetBeans, as with most Java-based applications, is easy. Just visit *netbeans.org* and click on *NetBeans IDE 5.0* under the *Latest Downloads* category at the top-right corner of the page. You can choose installers for your platform, including Windows, Mac OS, Linux and Solaris.

Before installation, you'll need a JDK 1.4.2 or higher installed and configured for use at the command-line. NetBeans uses JDK tools like the *javac* compiler, so a JRE won't be enough. If you don't yet have a JDK, there are download options bundling the latest JDK with the IDE.

I personally prefer to click on the link below *Other distributions, sources and extras* after the download form, and download instead the *NetBeans IDE 5.0 Archive*, choosing the *.zip* format. After all,

netbeans.org

NetBeans IDE home page

NetBeans is a pure-Java application, so you can use the same archive to install it on any platform with a suitable JDK. Just pick a directory and unpack the archive, and NetBeans is ready to run.

Starting and customizing NetBeans

After installing/unpacking NetBeans, the folder *bin* below the IDE installation folder will contain platform-specific commands to start the IDE. Windows users will use the *netbeans.exe* file, while Linux users will use the *netbeans* file. The IDE will open with a welcome page (see **Figure 1**).

If you have used the archive instead of the native installer, you'll get a license agreement dialog on the IDE's first run. Don't worry; the Sun Public License (SPL) used by NetBeans is an OSI-approved open source software license.

At the top of the IDE window, you see the main menu and toolbar. If you don't like the big toolbar icons configured by default, righ-click any empty spot in the toolbar and choose the *Small Toolbar icons* menu entry.

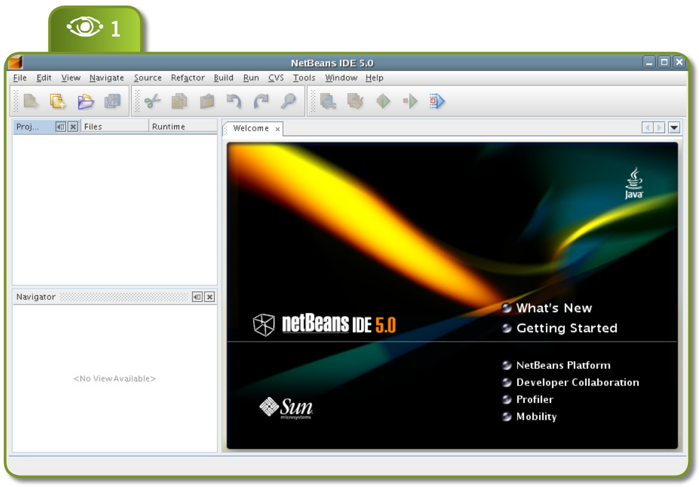
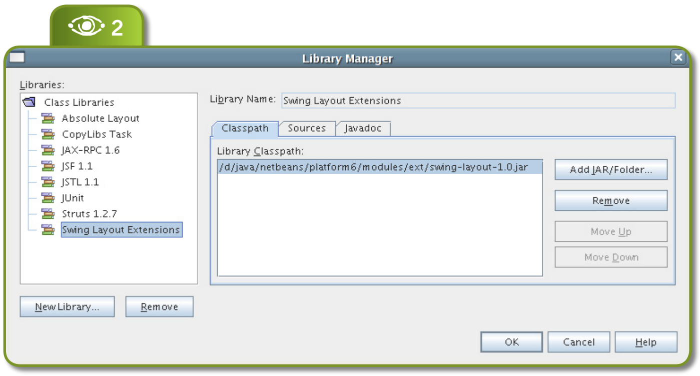


Figure 1 The NetBeans main window and welcome page



Library Name: Swing Layout Extensions

Library Classpath: /d/java/netbeans/platform6/modules/ext/swing-layout-1.0.jar

Figure 2 NetBeans Library Manager showing default libraries configured with NetBeans 5.0

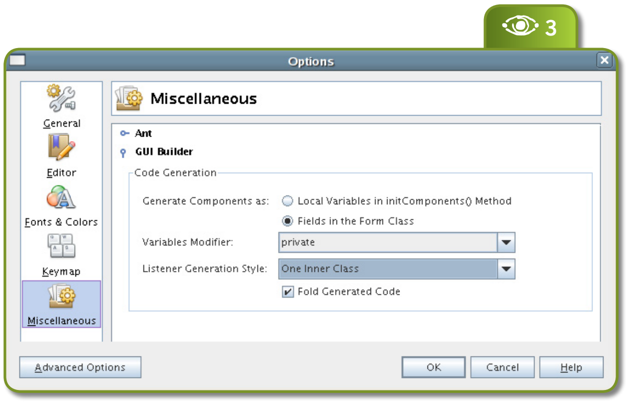
The left area contains two navigational panels. The top one is shared by the Projects, Files and Runtime windows. The bottom area contains the Navigator window, and the right-center area is used for the many editors included with NetBeans. Multiple editors and windows can share the same area; the IDE provides tabs for selecting the one displayed.

Most of the time, you'll use the Projects window to browse and edit Java code. The Navigator Window displays the structure of the artifact being edited; for example for Java code you'll see class attributes and methods, and for GUI design you'll see the component tree. The Files window is used when you need to see the physical file structure of your projects, and the Runtime window shows IDE processes and other environment resources like databases and Java EE servers.

To tweak your NetBeans environment, the two most used tools are the *Options Window* under *Tools|Options* on the main menu, and the *Library Manager* also under *Tools*. **Figure 2** shows the pre-configured libraries included with NetBeans 5.0, and **Figure 3** shows the first option I change before starting GUI development: the idiom for the code generated for Swing event listeners.

Most Swing tutorials and samples from books use anonymous inner classes (which is the installation default for NetBeans), but I find this idiom difficult to read and maintain. You get giant methods containing the code for handling many unrelated events. I prefer instead to have each listener as a named inner class, as shown in the figure.

The IDE provides a lot more customization than is shown by the Options window. Just click on the *Advanced Options* button and



Miscellaneous

- Ant
- GUI Builder

Code Generation

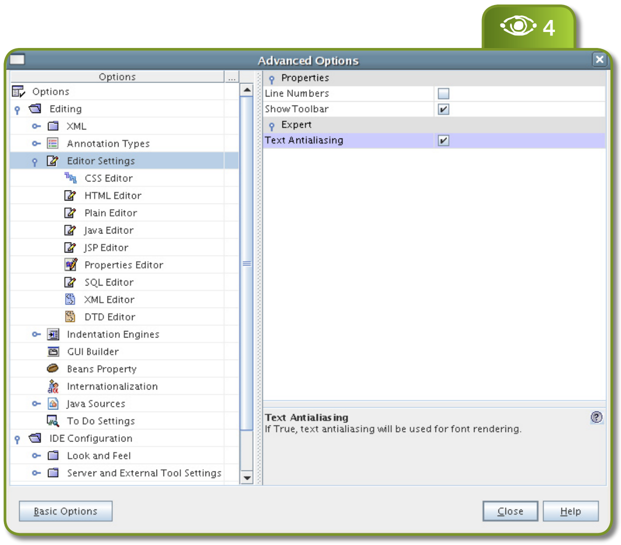
Generate Components as: ( ) Local Variables in initComponents() Method [✓ Fields in the Form Class]

Variables Modifier: private

Listener Generation Style: One Inner Class

✓ Fold Generated Code

Figure 3 NetBeans Options window: changing the default code-generation style for event listeners



```
Options
  Editing
    XML
    Annotation Types
    Editor Settings
      CSS Editor
      HTML Editor
      Plain Editor
      Java Editor
      JSP Editor
      Properties Editor
      SQL Editor
      XML Editor
      DTD Editor
    Indentation Engines
    GUI Builder
    Beans Property
    Internationalization
    Java Sources
    To Do Settings
  IDE Configuration
    Look and Feel
    Server and External Tool Settings
```

```
Properties
Line Numbers        [ ]
Show Toolbar        [✓]
Expert
Text Antialiasing   [✓]
```

Text Antialiasing
If True, text antialiasing will be used for font rendering.

Figure 4 NetBeans Advanced Options window, enabling text anti-aliasing for code editors.

you'll be presented with an expanded options dialog, as shown in **Figure 4**. This time you get a tree structure with hundreds of options grouped into categories. Most developers will want to enable anti-aliased text rendering on the editors, as shown by the figure, because this makes the code more readable.

Developing the sample app

Now that you had your first try at NetBeans 5, let's see the sample application we'll develop in the rest of this article. Here is a short list of requirements for it:

- Tasks should have a priority, so users can focus first on higher-priority tasks;
- Tasks should have a due date, so users can instead focus on tasks with are closer to their deadline;
- There should be visual cues for tasks that are either late or near their deadlines;
- Tasks can be marked as completed, but this doesn't mean they have to be deleted or hidden.

Most applications will have longer lists of requirements, and implementing even these simple example is not a trivial task. Building prototypes of the application UI helps end-users to state their requirements, and that's one of the reasons visual GUI builders became so popular. But even with a GUI builder, a quick sketch on paper can be of great help. We plan two main windows for the Todo application: a tasks list and a task-editing form. A rough sketch for both is shown in **Figure 5**.

After building the initial user interface prototype, it's important to show end-users a kind of functional prototype, which helps discuss the dynamics of user interaction in the application and the basic business process involved (if you are developing an Information System). This functional prototype reacts to user input but won't persist data.

That's where Object-Oriented development helps, because it's easy to create an initial set of objects the prototype can manipulate, and you can go very far developing the code to show and change

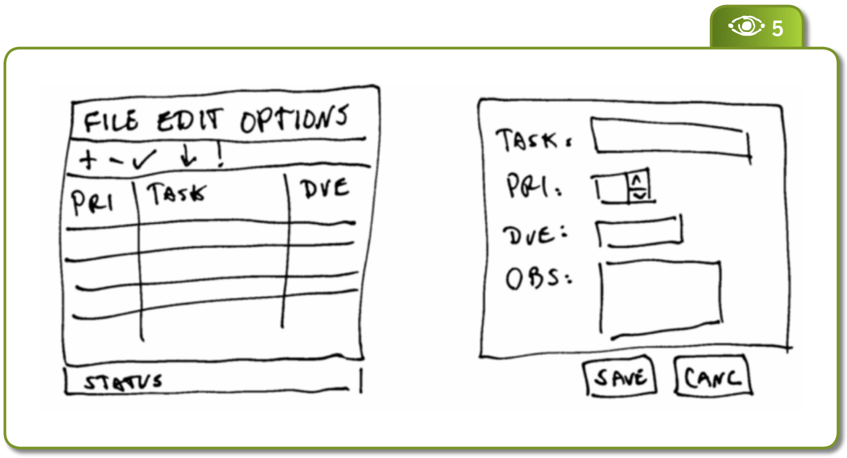


Figure 5 A sketch for the Todo user interface

NetBeans plug-in catalog
netbeans.org/catalogue

these objects without the need of persistent storage. Only when you have your functional specs and their implementation well advanced, at the end of the development process, do you need to worry about file formats or databases.

In my experience, this two-level prototyping approach improves productivity, and mixes well with TDD and other Extreme Programming practices, while keeping costs and schedule under control and meeting user demands. That leads us to developing the Todo application in three steps:

1. Build a "static" visual prototype of the user interface, using a visual GUI builder.

2. Build a "dynamic" prototype of the application, coding user interface events and associated business logic, and creating customized UI components as needed.

3. Code the persistence logic.

Designing the tasks list window

Let's go back to NetBeans. Any work in the IDE is done inside a project. Click on the *New Project* toolbar icon and select the *Java Application* entry in the *General* Category. Use "Todo" as the project name and choose a suitable project location (anywhere in your hard disk). Then click *Finish*.

NetBeans creates the project containing a Java package named after the project name and with a class named "Main" (in our case, **todo.Main**). Java conventions dictate that you should use your company DNS name as a prefix for all your package names, but in this example we'll waive that to keep things simple.

Now right-click the **todo** package icon and choose *New JFrame Form* (or choose *File|New File* from the main menu, and then select the *JFrame Form* from the *Java GUI Forms* category). Type "TasksWindow" as the class name. Notice that the IDE opens the visual form editor, as shown in **Figure 6**; notice also the location of the Projects, Navigator and Properties windows, the Palette and the editor area.

An orange frame highlights the selected component (the **JFrame** content pane in the figure). The navigator displays all visual and non-visual components on the **JFrame**, which is handy when you need to change the properties of a component hidden by another or too small to be selected in the drawing area.

To the right there's a component palette, which shows by default the standard Swing components (you can also add third-party JavaBeans), as well as the properties windows. Properties are categorized to ease access to the ones most commonly used, and changed properties have their names highlighted in bold.

To change the visual editor IDE layout, you can drag each window to another corner of the main window or even leave some windows floating around.

The NetBeans 5 visual editor is unlike other visual Java editors you may have seen. Just click right-click inside the **JFrame** and select the *Set Layout* menu item. You'll see the default choice is not a traditional Swing/AWT layout manager; it's something named "Free Design". This means you are using the Matisse visual GUI builder, one of the highlights of NetBeans 5.

Matisse configures the **JFrame** to use the **GroupLayout** layout manager developed in the SwingLabs java.net project, which will be included as a standard layout manager

6

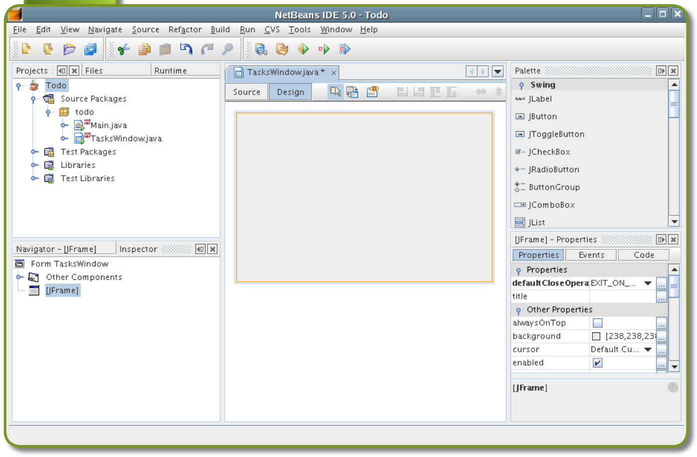


Figure 6
Visual editor with an empty JFrame

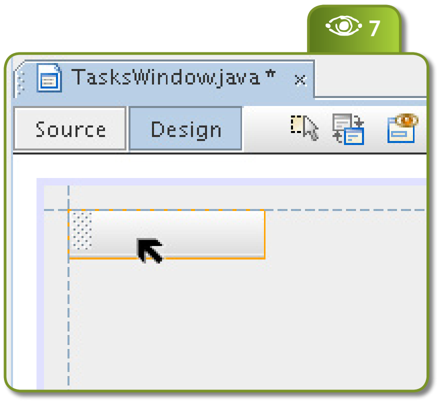


Figure 7
Visual guidelines help positioning and sizing controls in the visual editor

in Java SE 6.0. (You can learn more about **GroupLayout** in an article in this edition.)

If you choose any other layout, Matisse will be disabled and you will have the old NetBeans Visual Form editor. But of course we'll use Matisse, and you'll see how it brings a whole new meaning to "Swing visual design".

Select the Toolbar icon on the palette and move the mouse over the drawing area. You'll notice that a placeholder for the toolbar follows the mouse pointer, and that the visual editor displays guidelines when it's close to the edges of the **JFrame**, as shown in **Figure 7**.

These guidelines help you keep controls aligned and spaced out inside the container. Matisse generates the layout constraints to maintain the positioning of each component when the container is resized or when the Look and Feel (LAF) is changed. You design like you were in a free-form drawing area, but won't loose any of the advantages of using a Java layout manager.

As toolbars are usually attached to the window borders, move our toolbar to the top left corner of the **JFrame** (another set of guidelines will provide visual feedback helping component placement). Click to attach the toolbar at the desired location, and drag the right border so it becomes attached to the right **JFrame** border. **Figure 8** illustrates this process.

Repeat the process to insert a **JLabel** attached to the left, bottom and right corners of the **JFrame**. This label will be used as a status message area for the tasks window. Then insert a **JScrollPane**, attaching it to the left and right corners of the **JFrame** and to the bottom of the **JToolbar** and top of the **JLabel**. Just leave some spacing between the **JScrollPane** and **JFrame** borders, the **JToolbar** and the **JLabel**. The result so far should look like **Figure 9**.

Now try resizing the **JFrame** content panel (the drawing area). The **JToolbar**, **JLabel** and **JScrollPane** should resize to keep the borders attached to the **JFrame**'s corners and to each of the other borders.

Icons and Menus

By now you should have a pretty good idea about how to use NetBeans 5 to rapidly design a Swing UI. After adding buttons to the **JToolbar** and a **JTable** to the **JScrollPane**, the **TasksWindow** class will start to resemble the sketch we saw earlier. **JLabel**s are used as separators between each group of **JButton**s inside the **JToolbar** (the Swing **JSeparator** won't behave as expected). Later on we'll customize the **JTable**.

NetBeans doesn't come with a good set of icons for use by

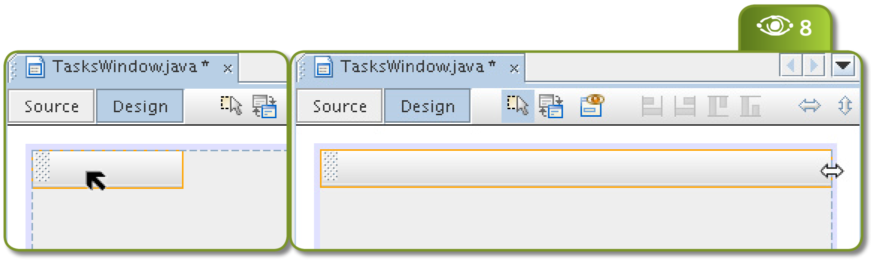


Figure 8
Positioning and resizing the toolbar so it is attached to the left, top and right corners of the JFrame. Notice the guidelines over the drawing borders

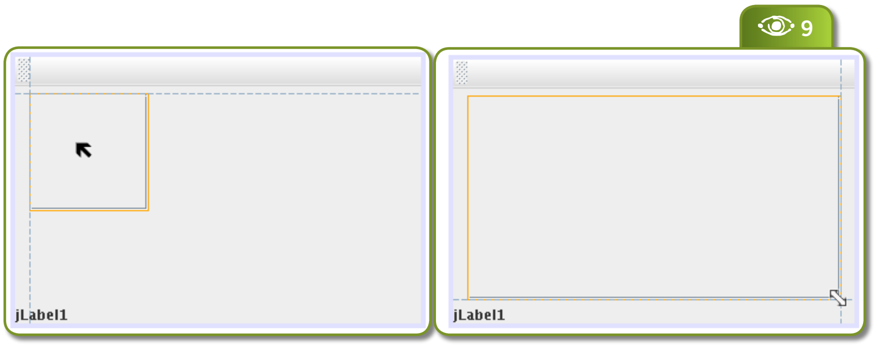
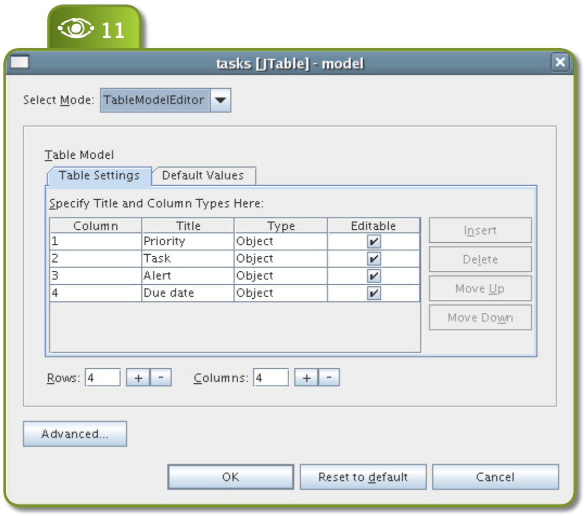


Figure 9
Positioning the JLabel and the JScrollPane



tasks [JTable] - model

Select Mode: TableModelEditor

Table Model

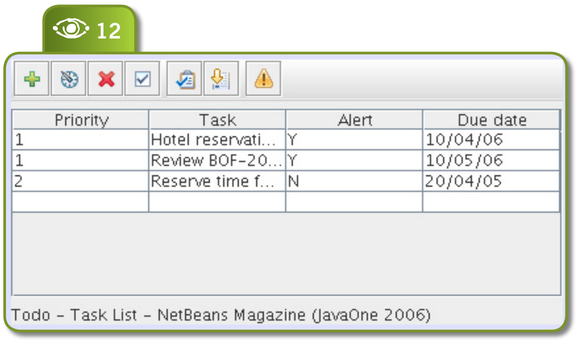
Table Settings | Default Values

Specify Title and Column Types Here:

Column	Title	Type	Editable
1	Priority	Object	✓
2	Task	Object	✓
3	Alert	Object	✓
4	Due date	Object	✓

Rows: 4 Columns: 4

Figure 11 JTable model customizer after adding a few columns



Priority	Task	Alert	Due date
1	Hotel reservati...	Y	10/04/06
1	Review BOF-20...	Y	10/05/06
2	Reserve time f...	N	20/04/05

Todo – Task List – NetBeans Magazine (JavaOne 2006)

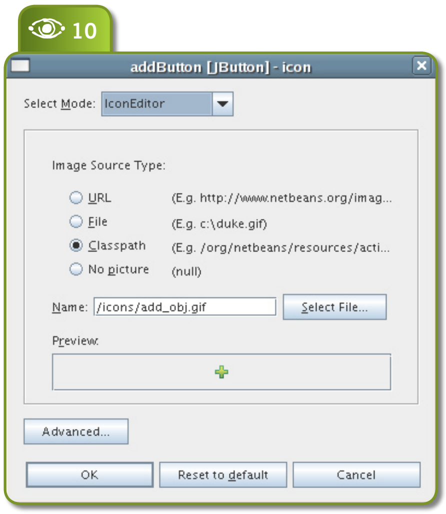
Figure 12 TasksWindow UI so far

applications, so I borrowed some from a few other open source projects; they are provided together with the sources for this article (see the URL at the last page). But instead of referring to the icons by file system paths, which would lead to additional logic to find the icon files on the user machine, I created an *icons* folder under the project *src* folder (which corresponds to the *Source Packages* item in the Projects window) and copied all icon files there. The NetBeans Projects window will display non-Java files inside the source folder, so you won't need to switch to the Files window just to see them.

Non-Java files inside the *Source Package* folder will be added by NetBeans to the application jar file, and the application will be able to refer to them as classpath resources, no matter where the jar file is installed on the user machine. As a bonus, the application code doesn't have to worry about platform-specific details like path separators and drive letters.

NetBeans provides a customizer for editing a component's icon property. Click on the ellipsis button at the right side of the **icon** property to open the customizer shown in **Figure 10**. Use this customizer to configure the icon for the component. After selecting the *Classpath* radio button, the *Select File* button will allow you to browse the icon files and select interactively the one to use (including icon files inside jar packages).

NetBeans also provide a customizer for **JTable**'s **model** property; see **Figure 11**. We'll use it to make the **JTable** appear in the visual editor with correct column names and some sample data. This is not the way the final application will look like. Typically, the customization of **JTable**s will require the development of custom Java classes like cell renderers and column models – because **JTable**s are views for Java objects, not just for plain Strings like in other GUI toolkits.



addButton [JButton] - icon

Select Mode: IconEditor

Image Source Type:

- ○ URL (E.g. http://www.netbeans.org/imag...
- ○ File (E.g. c:\duke.gif)
- ● Classpath (E.g. /org/netbeans/resources/acti...
- ○ No picture (null)

Name: /icons/add_obj.gif Select File...

Preview:

Figure 10 NetBeans customizer for the icon property Library Manager, showing default libraries configured with NetBeans 5.0

But even if the customization done using the NetBeans visual editor won't be used in the final application, it's useful to do that work. A prototype should display actual data (or at least data a typical user would input, instead of placeholder text). This will help users understand the prototype and make sure the UI allows for sufficient space for displaying the actual data.

Now the prototype should look like **Figure 12**. The main element still missing is the application menu bar. To add it, select the **JMenuBar** control on the palette and click anywhere inside the drawing area (except inside the **JToolbar** or its **JButton**s). To edit the menu bar, you don't use the component palette or Matisse features. Just open the **JMenuBar** context menu (right-click) and choose *Add JMenu*. Then you can select the new **JMenu** and configure its properties. For the Todo application, we need to add menus with labels "File", "Edit", "Options" and "Help".

Adding menu items follows a similar procedure. You use the context menu for the **JMenu** and open the submenu *Add* to select between **JMenuItem**, **JCheckBoxMenuItem**, **JRadioButtonMenuItem**, **JMenu** and **JSeparator**, as shown in **Figure 13**.

The added menu items won't be shown in the drawing area, so they can't be selected directly for customization. But the Navigator window allows access to the items, and the Properties window reacts to selection on the Navigator the same way it does in the drawing area. **Figure 14** shows all menus to guide you in completing the **TasksWindow**.

In the last figure you may have noticed underlined characters (like the "F" in "File") and keyboard accelerators (like *Alt-x* for "Exit"). These are configured respectively by the **mnemonic** and **accelerator** properties.

The meaning of each menu item should be self-explanatory, given the application requirements and the fact we'll use a file-based database as persistent storage for tasks.

Designing the task details dialog

Now we'll create the **TaskDetailsDialog**. Right-click the **todo** Java package and select *New>File/Folder*. Then choose *JDialog Form* in the *Java GUI Forms* category. We start with a **JLabel** attached to the left, top and right borders of the dialog, with no spacing. It will serve as a message area for validation errors and the like. Set its **opaque** property and the **background** color so it looks like a band at the top of the dialog. Also add an **EmptyBorder** (**border** property) so there's empty space around the text and the band's borders.

Now add three **JLabel**s for the description, priority and due date fields. Attach all three to the left of the **JDialog** internal area (the drawing area). Leave some spacing between the components and the border. Resize the two shorter labels to attach their right borders to the right border of the larger one. **Figure 15** illustrates this procedure.

Then select the three labels (with shift + click) and change the **horizontalAlignment** property to **RIGHT**. After that, insert a **JTextField**, a **JSpinner** and a **JFormattedTextField** at the left of each label. Note that the guidelines keep the label and text field baseline aligned, as shown in **Figure 16**.

The **JSpinner** does not provide any property to set a preferred or minimum width, while the **JTextField** and **JFormattedTextField** use the **column** property for this. But you can resize the **JSpinner** and Matisse will set the component's preferred size in pixels.

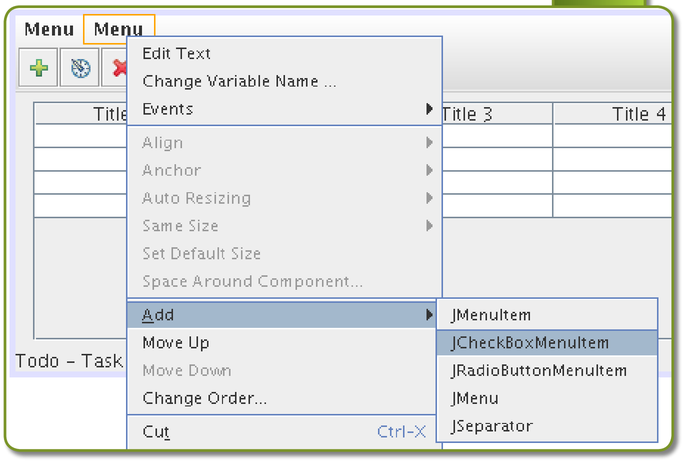


Figure 13
Adding items do a JMenu inside a JMenuBar

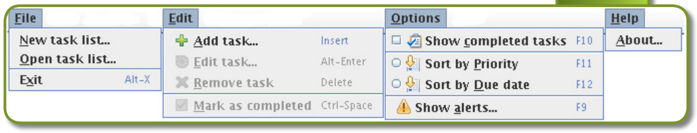


Figure 14
Complete menus for the TasksWindow JFrame

Sizing GUI components in pixels is not guaranteed to work well in different platforms, or if your users change the default Swing LAF. Use this Matisse feature with care!

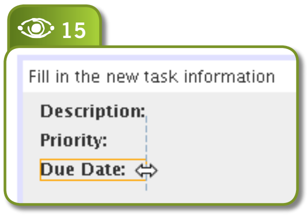


Figure 15 Sizing the TaskDetailsDialog labels

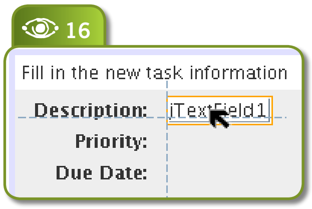


Figure 16 The "free design" layout aligns different kinds of controls by the text baseline

By now you should not have problems positioning and aligning the remaining components in the **TaskDetailsDialog**. **Figure 17** shows its final appearance as a reference to the reader.

Good UI design makes all buttons from a logical group the same size, and Matisse can enforce this good practice. Just select all desired buttons (actually you can select any control you want) and right-click any of the selected buttons. Then check the *Same Size | Same Width* checkbox menu item. The drawing area will indicate that the controls were configured to always have the same size, as shown in **Figure 18**.

Deploying the first prototype

To finish the prototype, the **todo.Main** class needs code to create the **TasksWindow** and make it visible. Besides, there should be an Action listener in any toolbar button or menu item to show the **TaskDetailDialog**. The code is shown in **Listing 1**. The reader should be able to fill the missing pieces, like **package** and **import** statements.

You can then use command *Run | Run Main Project* from the main menu, the toolbar button or press *F6* to run the prototype. After checking that it looks as intended (it does nothing besides displaying the two windows) you can use either the menu item *Build | Build Main Project* or *F11* to create an executable jar that can be deployed on end user machines for prototype validation.

The jar package is created in the *dist* project folder. You can verify this in the Files window. NetBeans also creates the folder *dist/lib* containing any libraries used by the application, and configures the jar manifest to point to the project's main class and libraries. **Figure 19** presents the *dist* project folder as seen in the NetBeans Files window, and **Listing 2** shows the manifest file generated for the *Todo.jar* archive.

Note the library *swing-layout-1.0.jar* inside the *dist/lib* folder. This contains the GroupLayout used by UIs built with Matisse.

So all you need to do is copy the contents of the *dist* folder to the user machine (you don't need to preserve the file name *dist*), and then run **java -jar Todo.jar**.

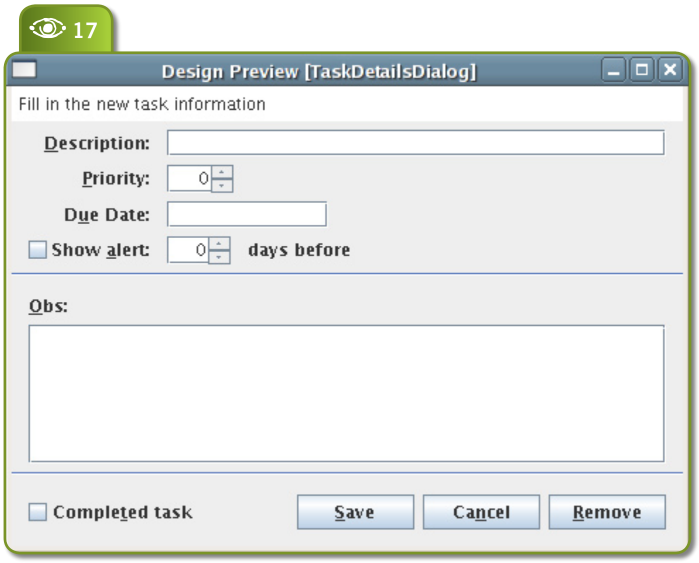


Figure 17 Finished prototype for the TaskDetailsDialog

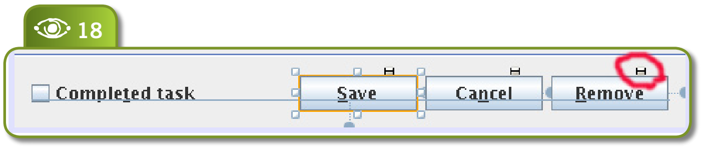


Figure 18 The IDE's visual hint for same-size components

End of Step 1

This prototype is almost the finished application from the UI design perspective, but in real projects you shouldn't spend too much time perfecting its looks. Remember, the prototype is a tool to gather and validate user requirements and lessen the risk of missing important application functionality.

The problem is the user often cannot un-

19

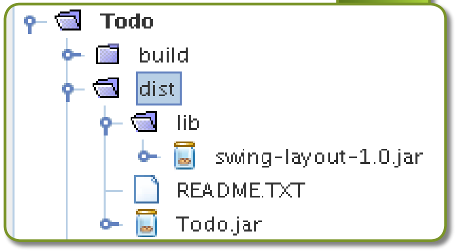


Figure 19 Distributable files for deploying the prototype to end-users

derstand that an "almost ready" UI is not an "almost finished" application. That makes many developers avoid prototyping during development.

An interesting approach to mitigate this problem is to use a custom look-and-feel that makes the application look unfinished. The Napkin Project at SourceForge (*napkinlaf.sf.net*) provides such a LAF. See the **sidebar** "Using the Napkin LAF in a NetBeans project" for details.

Listing 1. Code to finish the first prototype.

todo.Main (todo/Main.java)

```
(...)
 public static void main(String[] args) {
    JFrame w = new TasksWindow();
    w.pack();
    w.setVisible(true);
}
```

todo.view.TasksWindow (todo/view/TasksWindow.java)

```
(...)
private void addButtonActionPerformed (
    java.awt.event.ActionEvent evt) {
  JDialog d = new TaskDetailsDialog(this, true);
  d.pack();
  d.setVisible(true);
}
```

Listing 2. jar archive manifest file (META-INF/MANIFEST.MF) generated by NetBeans

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.6.5
Created-By: 1.5.0_05-b05 (Sun Microsystems Inc.)
Main-Class: todo.Main
Class-Path: lib/swing-layout-1.0.jar
X-COMMENT: Main-Class will be added automatically by build
```

Using the Napkin LAF in a NetBeans project

The Napkin Look-and-Feel can give an important "unfinished" look to your prototype. Follow these steps to use it in the example application:

1. Visit the Napkin home page (*napkinlaf.sf.net*), click on the link next to "latest release" and download the archive *napkinlaf-version.zip;* unpack the zip to a folder of your choice;

2. Right-click the *Todo* project icon on the NetBeans Project window and select the *Properties* menu item.

3. Select the *Libraries* category and click the button *Add JAR/Folder*, and browse for the archive *napkinlaf.jar* inside the folder where you unpacked the Napkin download.

4. Rebuild the application so the *napkinlaf.jar* archive gets copied to the *dist/lib* folder and the jar manifest gets updated with a reference to the new library.

5. Add the following code to the start of the main method:

```
UIManager.setLookAndFeel(
  "net.sourceforge.napkinlaf.NapkinLookAndFeel");
```

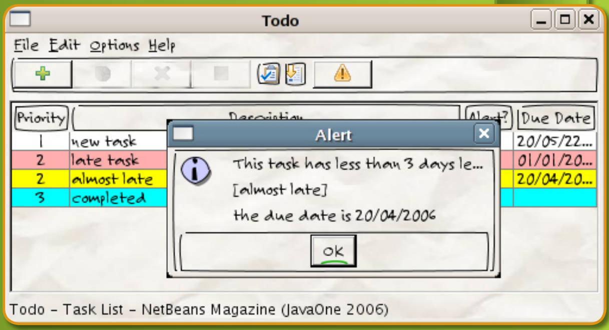
As an alternative, include the command-line option **-Dswing.defaultlaf=net.sourceforge.napkinlaf.NapkinLookAndFeel** when starting the application.

Figure S1 shows the look of the Todo app using the Napkin LAF.

napkinlaf.sf.net

Napkin custom Swing LAF home page

S1

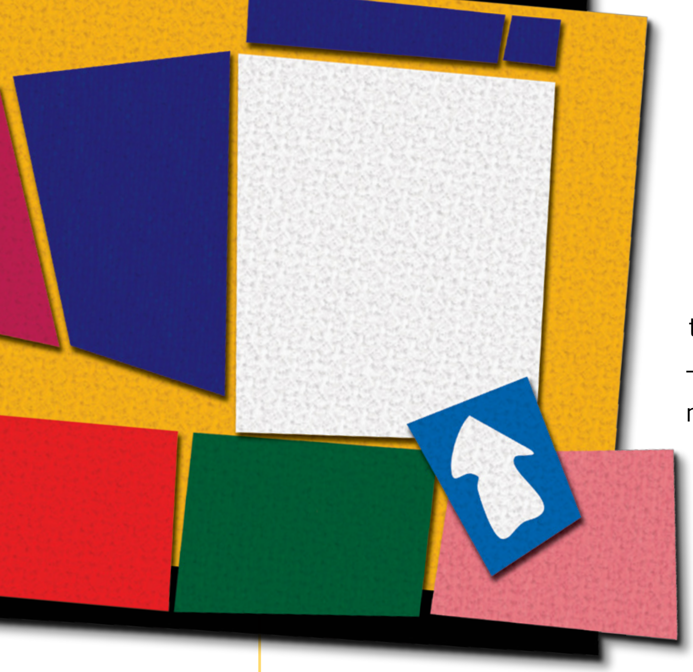


Priority	Description	Alert?	Due Date
1	new task		20/05/22...
2	late task		01/01/20...
2	almost late		20/04/20...
3	completed		

This task has less than 3 days le...
[almost late]
the due date is 20/04/2006

Todo - Task List - NetBeans Magazine (JavaOne 2006)

Figure S1 Sketch for the Todo user interface



Todo Applicaton Architecture

The second step – building the "dynamic prototype" – aims to implement as much user interaction as possible without using a persistent storage or implementing complex business logic. It's well known that in most GUI applications the event-handling code accounts for above 60% of the total application code. Therefore it pays off to develop the UI code incrementally. Another reason to build this second prototype is most users have trouble imagining how an application should behave if there's nothing they can click and see results in.

This can be achieved by creating "mock" objects for business logic and data access. Such objects should have the same public interface as the real objects (and will help validate the intended interface before it's implemented and changes become too expensive), but will return hard-coded data. You can use the Java collections API so the user can change an object and see the changes until the application is closed.

If you think of the **TasksWindow** as a black box that can display a collection of task objects, and of the **TaskDetailDialog** as a black box capable of displaying a single task object, it's not hard to think in terms of mock objects.

We'll use two well-known design patterns in the Todo application: DAO (Data Access Object) and the MVC (Model-View Controller). We'll also define a VO (Value Object) named **Task** for moving information between application tiers. Therefore the view classes (such as the **TasksWindow** and **TaskDetailsDialog**) will receive and return either **Task** objects or collections of **Task** objects. The controller classes will transfer those VOs from view classes to model classes, and back.

Figure 20 shows a UML class diagram for the main application classes. Methods and attributes were omitted, but we'll describe the most important ones. The full sources

NetBeans 5.5 preview, featuring UML CASE features

netbeans.org/community/releases/55/

20

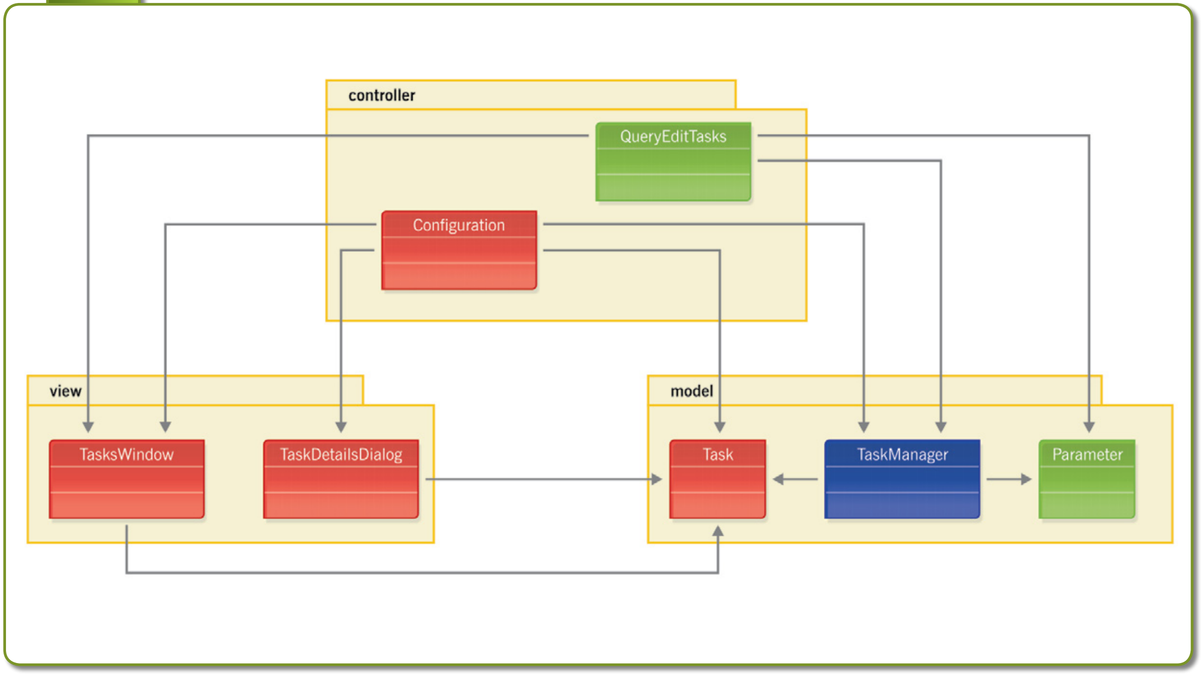


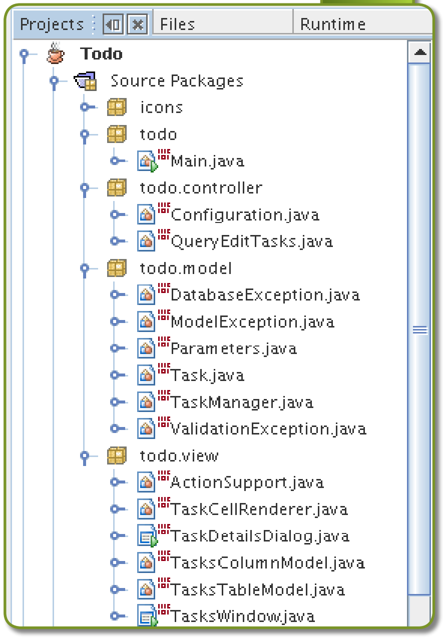
Figure 20 UML class diagram for the main application classes

for the finished application are available for download; see the link at the end of the article.

This UML model was drawn using ArgoUML (*argouml.org*) a powerful open source software CASE tool written in Java. Developers interested in CASE tools can also check NetBeans 5.5 currently in preview, which includes UML tools originally developed as part of Sun Java Studio Enterprise.

Now we'll create Java packages corresponding to the MVC class roles: **todo.view**, **todo.controller** and **todo.model**. Create these packages by right-clicking the *Source Packages* icon on the Projects window, and selecting *New|Java Package*. Then move the classes already created for the first prototype to the **todo.view** package by right-clicking each one and selecting *Refactor|Move Class*. While nothing stops you from using the Java code editor to change the **package** statement for each class (and even the class name), using the *Refactor* menu automatically changes all references in other classes.

21



```
Todo
  Source Packages
    icons
    todo
      Main.java
    todo.controller
      Configuration.java
      QueryEditTasks.java
    todo.model
      DatabaseException.java
      ModelException.java
      Parameters.java
      Task.java
      TaskManager.java
      ValidationException.java
    todo.view
      ActionSupport.java
      TaskCellRenderer.java
      TaskDetailsDialog.java
      TasksColumnModel.java
      TasksTableModel.java
      TasksWindow.java
```

The finished application will contain more classes, some of which can be considered "internal" to their respective packages. Others play just an accessory role, such as exception classes. **Figure 21** shows all classes from the finished application in the NetBeans Projects window.

Here's the plan for building the second prototype:

1. Adjust column widths for the tasks list and display the visual cues for late and completed tasks;

2. Handle selection events to enable and disable menu and toolbar items;

3. Handle action events to sort and filter the tasks list;

4. Handle action events to create, edit and remove tasks.

Items 1 to 3 can be implemented and tested with a mock model object (**TaskManager**) that always returns the same task collection. Item 4 can be tested with a mock object that simply adds or removes objects from that collection.

Customizing a JTable

In order to customize the Swing **JTable** so it displays a collection of **Task** objects, we provide adequate column widths for each column in the task list (which corresponds to **Task** attributes) and change each row background colors according to the task status: red for late tasks, yellow for tasks with an alert set, blue for completed tasks, and white otherwise.

Most task list columns have a short content and can have fixed width. Just the description column/attribute can have wider content, so it should get whatever space is left after the fixed-width columns are sized. To implement all these features, we need to create three classes:

▪ The **TasksTableModel** class receives requests for data at a specific row/column from the **JTable** control and returns a task attribute value, such as the description or due date, for the task at the given row. It also has the ability to filter the task collection to exclude completed tasks, or to change the sorting criteria. Sometimes it will be better to leave these sorting and filtering tasks to the model (which can delegate them to the database) but if the dataset is not too big, doing these operations in-memory will improve user experience.

▪ The **TaskColumnModel** class adds columns to the **JTable** and configures each column with its preferred width, label and

argouml.org

ArgoUML, Open Source UML CASE tool written in Java

Figure 21
All classes for the Todo application

resizeability. It completely replaces at runtime the **DefaultTableModel** created and configured by the NetBeans visual editor.

- **TaskCellRenderer** provides a Swing **JLabel** configured with the correct background color for the task being shown. It also formats the **dueDate** and **completed** attributes as strings.

Figure 22 shows the final appearance of the **TasksWindow**, with the customized **JTable**.

Handling Internal Events

Having a display of tasks ready, it's time to add some event-handling. It will be useful to separate UI events into two mutually exclusive categories:

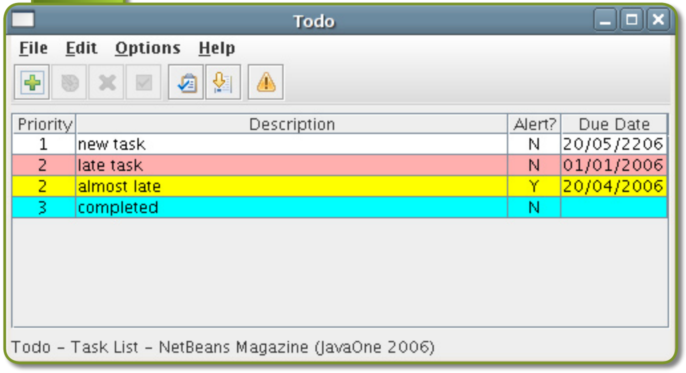
1. Internal events, that affect just the view itself.
2. External events, which cause model methods to execute.

Among internal events, are selection changes and clicks on Cancel buttons. These are handled by the view classes themselves, and are not exposed as part of the view classes' public interfaces. For example, the selection of a task should enable the *Edit task* and *Remove task* menu item, and the corresponding toolbar buttons.

Swing trail of the Java Tutorial
java.sun.com/docs/books/tutorial/uiswing

Swing itself was designed using the MVC architecture, but don't be confused: Swing "model" classes have nothing to do with application "model" classes. Don't make your model classes implement Swing model interfaces, and don't make them subclasses of Swing classes. From the application architecture point of view, Swing model classes are part of the *view* tier if the application uses MVC.

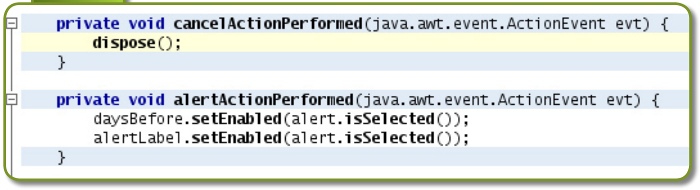
22



Priority	Description	Alert?	Due Date
1	new task	N	20/05/2206
2	late task	N	01/01/2006
2	almost late	Y	20/04/2006
3	completed	N	

Todo – Task List – NetBeans Magazine (JavaOne 2006)

23



```
private void cancelActionPerformed(java.awt.event.ActionEvent evt) {
    dispose();
}

private void alertActionPerformed(java.awt.event.ActionEvent evt) {
    daysBefore.setEnabled(alert.isSelected());
    alertLabel.setEnabled(alert.isSelected());
}
```

Figure 23 Event handler declarations are guarded (i.e. non-editable) code sections in the source editor

To code an internal event handler, either right-click on the control that is the event source and select the desired event from the *Events* sub menu, or select the *Events* category from the Properties window. Both the context menu and the Property window will highlight in boldface the events that already have a handler.

You can change the implementation of an event-handler method, but you cannot remove the method or change its name in the source editor. To do this, you have to use the Properties window. **Figure 23** shows some event handlers in the source editor.

Handling External Events

The category of events we call "external" should not be handled by view classes. They should instead be forwarded to controller classes, which usually implement the workflow logic for a specific use case or a related set of use cases.

To help with this, the application includes the **todo.view.ActionSupport** class. This class simply keeps a list of **ActionListener**s and forwards **ActionEvent**s to them. But **ActionSupport** is itself an **ActionListener**. This is done to avoid having lots of event-related methods, e.g. **add/removeNewTaskListener()**, **add/removeEditTaskListener()** and so on. Instead, view classes generate only an **ActionEvent**. The **ActionSupport** classes capture **ActionEvents** from the view components and forward them to the controller, which registers itself as a view **ActionListener**.

However, if the same **ActionListener** inside the controller class receives **ActionEvents** originated from multiple sources inside a view class, how can the controller know which operation is being performed by the user? The "secret" is the **actionCommand** attribute from the **ActionEvent**, which is initialized from the **actionCommand** property from the source component. So the implementations of the controller classes are basically a sequence of **if/else if** statements checking for the **actionCommand** string.

Many developers balk at this idea, claiming this is not an "object-oriented" way of doing things. But nothing prevents you to create to a generic controller framework, where the event dispatch information comes from an XML configuration file and/or is handled by an IoC controller.

End of Step 2

Now that we have fully functional view and model classes, it's time to start replacing the mock implementations of the model classes by real logic using persistent storage.

In large application projects, you could have a team working on the UI, building the two prototypes in sequence as we did, and another team working on business and persistence logic, preferably using TDD. They can work in parallel and join at the end, putting together functional view and controller implementations with functional model implementations.

Most of the work in this step was just coding. NetBeans provides nice code editors and a good debugger that eases the task providing the usual benefits: code-completion, JavaDoc integration and refactoring support. But it can go beyond: it's very easy to build in NetBeans 5 new plug-in modules to package your project coding standards, such as project templates, controller class templates and so on.

Model classes

The **TaskManager** class is a DAO (Data Access Object). Being the only DAO on the application, it contains many methods that would otherwise be in an abstract superclass. Its implementation is very simple, so there's lots of room for improvement.

There's another model class: **Parameter**. It uses the Java SE Preferences API to store configuration data such as the path to the current tasks database. A desktop application should be as plug-and-play as possible, so the application will initialize a default tasks database if there isn't one available. But it's flexible enough to allow the user to open task databases at other locations, and remember the last one used.

The Todo application uses HSQLDB (*hsqdb.org*), an embedded Java database. This allows the application to meet the ease-of-deployment requirements for a typical desktop application. You just need to download HSQLDB and add the archive *hsqldb.jar* to the NetBeans project libraries.

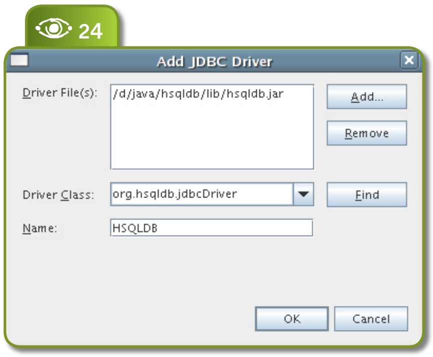
hsqldb.org

HSQLDB, an Open Source embedded 100%-Java database

Inspecting the Database

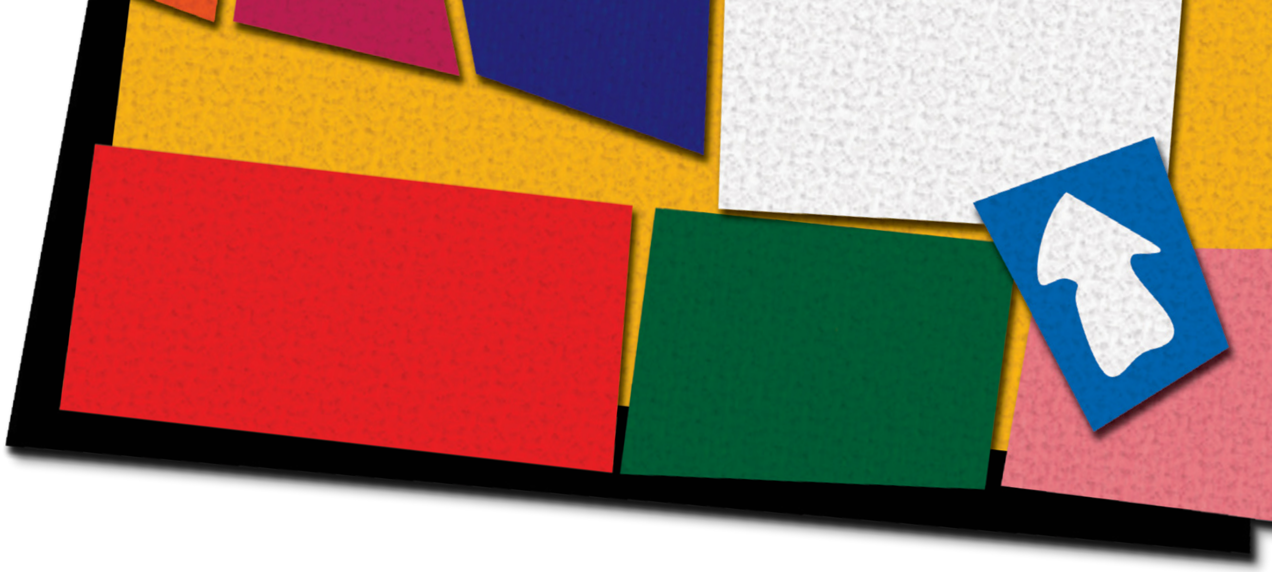
When developing and debugging persistence code, developers usually need a way to tap into the database. Maybe they need to check the effect of an update, or change some table definition. NetBeans provides direct support for browsing any JDBC-compliant database and submit SQL commands.

Switch to the Runtime window (it is normally hidden by the Projects and Files windows) or open it from the *Window* menu. Expand the *Databases* and then the *Drivers* categories. Right-click on the Drivers icon, and select *Add Driver*. Fill the dialog's fields with the location of your *hsqldb.jar* archive, as shown in **Figure 24**. NetBeans will often set the database driver class name by itself.



Driver File(s): /d/java/hsqldb/lib/hsqldb.jar
Driver Class: org.hsqldb.jdbcDriver
Name: HSQLDB

Figure 24 Configuring the HSQLDB JDBC driver in the IDE



Fernando Lozano (*fernando@lozano.eti.br*) is an independent consultant with more than 10 years experience doing IS development and network integration. He has many professional certifications, and is also a technical writer and open source software advocate, serving as Linux.java.net community manager, LPI Brazil Technical Director and GNU Project webmaster in charge of Brazillian Portuguese translations.

Now right-click the HSQLDB driver icon, and choose the *Connect using menu* item. Provide the parameters to connect to your local Todo database, using **Figure 25** as a template. The default database location is *db/todo* under the *{user.home}* folder, which is usually */home/user* under Linux or *C:\Documents And Settings\UserName* under Windows.

Then you can open the connection and browse the database catalog for tables, indexes and other database objects. Each item has a context menu for operations like creating new tables, altering columns and viewing data (**Figure 26**). Most operations have easy-to-use wizards.

The Todo application uses HSQLDB in the stand-alone mode, which locks the database files for exclusive access. So it won't be possible to use the NetBeans database console while the application is running. However it's possible to run HSQLDB in server mode accepting concurrent connections from multiple clients, allowing the inspection of a live task list database. Check the HSQLDB manual for instructions on how to start and connect to the database server.

End of Step 3 & Conclusions

The Todo application is completed. Although simple in scope and with only a few classes, it demonstrates many practices that could improve your desktop Java application quality and development speed.

Also shown were many features that NetBeans provides to increase developer productivity. NetBeans goes beyond visual development by supporting coding activities with specialized editors for Java, Ant, XML and other languages, besides CVS, JUnit and refactoring support and a database console.

netbeans.org/community/magazine/code/nb-completeapp.zip

25

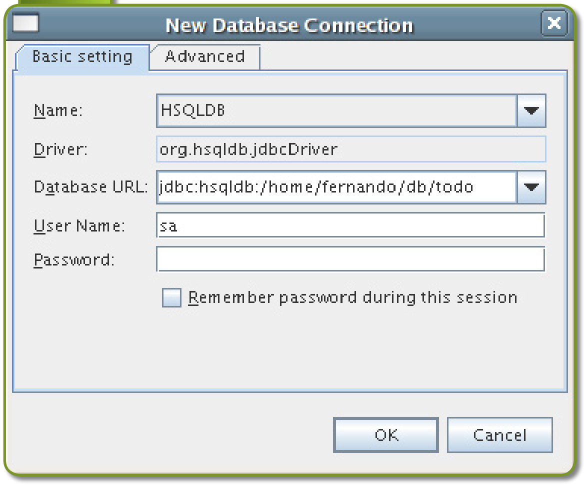


Figure 25 Connecting to the Todo task database

New Database Connection

Basic setting | Advanced

Name:	HSQLDB
Driver:	org.hsqldb.jdbcDriver
Database URL:	jdbc:hsqldb:/home/fernando/db/todo
User Name:	sa
Password:	

☐ Remember password during this session

OK Cancel

26

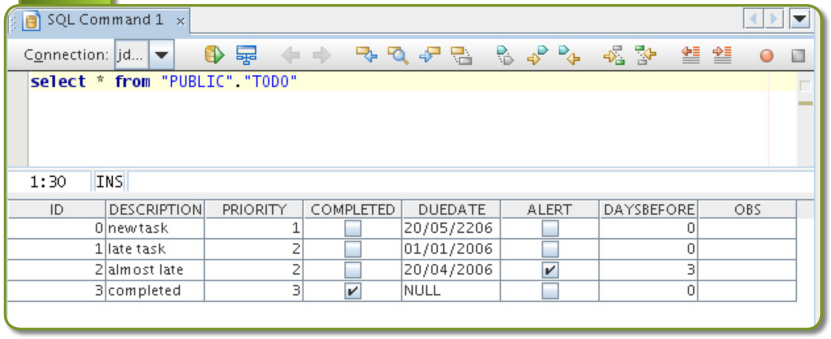


Figure 26 Executing SQL statements

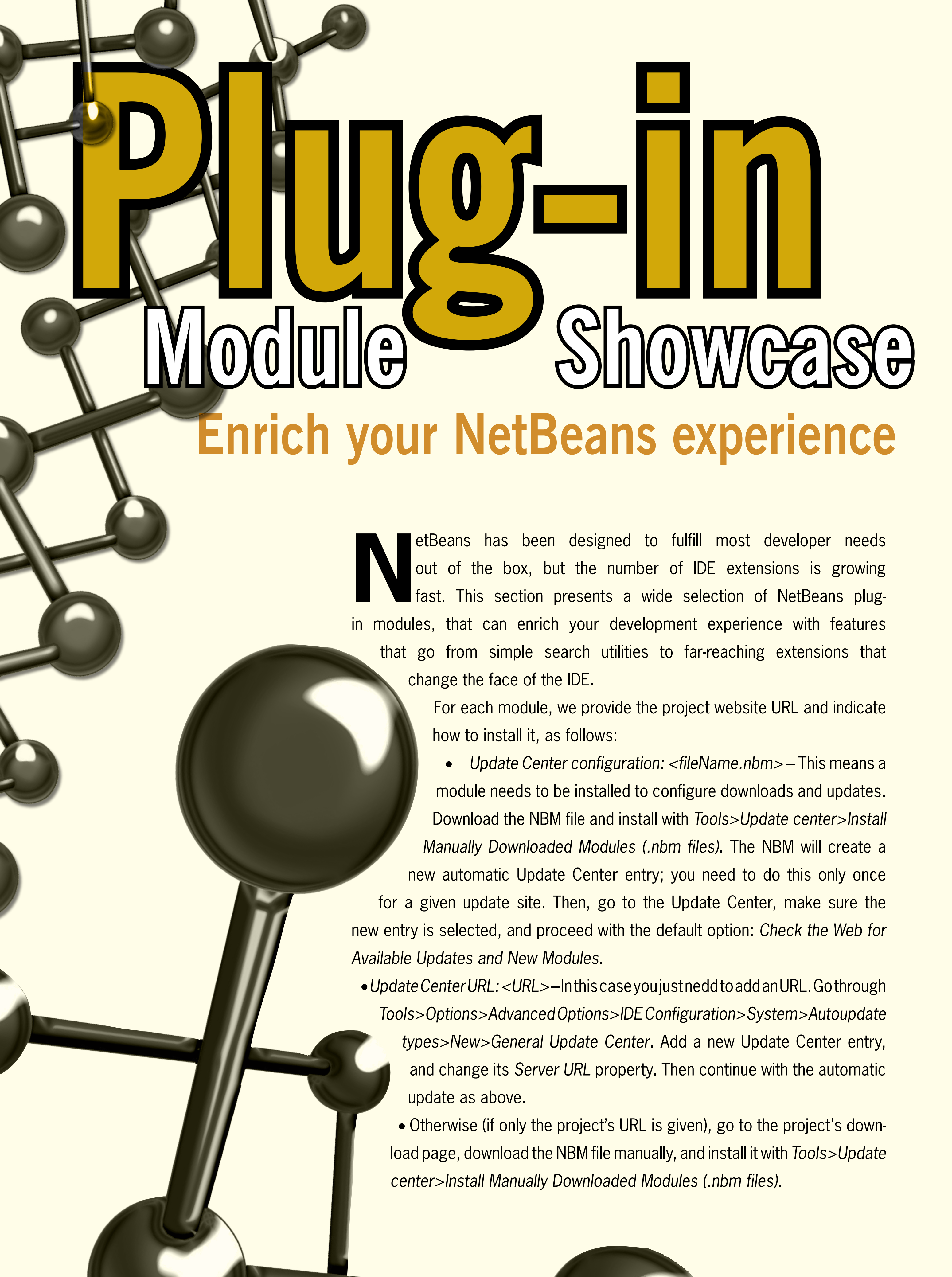
SQL Command 1

Connection: jd...

```
select * from "PUBLIC"."TODO"
```

1:30 INS

ID	DESCRIPTION	PRIORITY	COMPLETED	DUEDATE	ALERT	DAYSBEFORE	OBS
0	new task	1	☐	20/05/2206	☐	0	
1	late task	2	☐	01/01/2006	☐	0	
2	almost late	2	☐	20/04/2006	✓	3	
3	completed	3	✓	NULL	☐	0	



Plug-in Module Showcase

Enrich your NetBeans experience

NetBeans has been designed to fulfill most developer needs out of the box, but the number of IDE extensions is growing fast. This section presents a wide selection of NetBeans plug-in modules, that can enrich your development experience with features that go from simple search utilities to far-reaching extensions that change the face of the IDE.

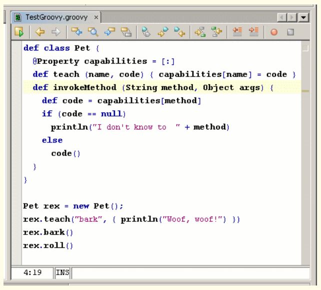
For each module, we provide the project website URL and indicate how to install it, as follows:

- *Update Center configuration: <fileName.nbm>* – This means a module needs to be installed to configure downloads and updates. Download the NBM file and install with *Tools>Update center>Install Manually Downloaded Modules (.nbm files)*. The NBM will create a new automatic Update Center entry; you need to do this only once for a given update site. Then, go to the Update Center, make sure the new entry is selected, and proceed with the default option: *Check the Web for Available Updates and New Modules*.
- *Update Center URL: <URL>* – In this case you just nedd to add an URL. Go through *Tools>Options>Advanced Options>IDE Configuration>System>Autoupdate types>New>General Update Center*. Add a new Update Center entry, and change its *Server URL* property. Then continue with the automatic update as above.
- Otherwise (if only the project's URL is given), go to the project's download page, download the NBM file manually, and install it with *Tools>Update center>Install Manually Downloaded Modules (.nbm files)*.

Coyote

coyote.dev.java.net

The Coyote Module provides support for dynamic and scripting languages for the Java SE platform. The current version supports both Groovy and Jython, but a common framework for other languages is planned. Some features provided are syntax coloring, script execution and support for unit tests written in Jython or Groovy.



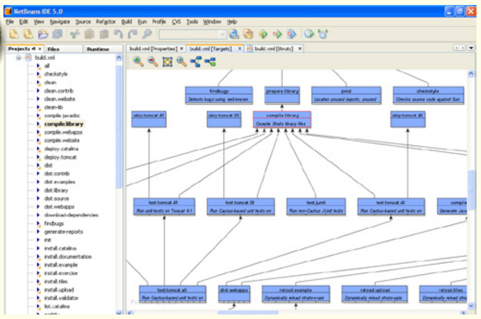
```
def class Pet {
  @Property capabilities = [:]
  def teach (name, code) { capabilities[name] = code }
  def invokeMethod (String method, Object args) {
    def code = capabilities[method]
    if (code == null)
      println("I don't know to  " + method)
    else
      code()
  }
}

Pet rex = new Pet();
rex.teach("bark", { println("Woof, woof!") })
rex.bark()
rex.roll()
```

Coyote: Groovy script editing and execution

Ant Explorer

yworks.com/en/products_antexplorer_netbeans.htm

yWorks AntExplorer supports visualization and execution of Ant buildfiles, providing graphs that show dependencies between targets and property definitions. Graphs are zoomable and it's possible to focus on a target, show its dependencies, and navigate to its source. To activate the plug-in, righ-click an Ant buildfile and select "Visualize".



Ant Explorer: exploring Ant buildfiles visually

Jackpot

jackpot.netbeans.org

The Jackpot project develops tools for code reengineering, which goes beyond standard refactoring addressing the needs of complex changes in large amounts of code. This is accomplished by user-definable transformation code or rules that can match and change code.

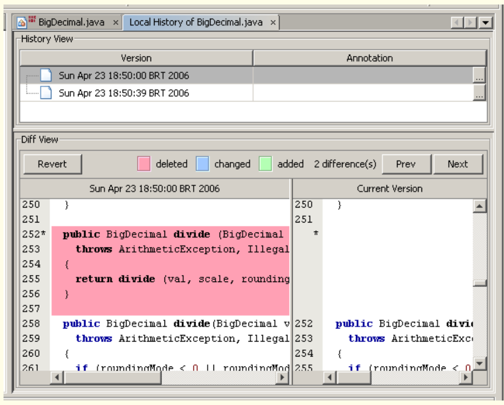
You could use Jackpot, for example, to remove all uses of a deprecated method, replacing them for equivalent code, even if the change is not as easy as replacing method names. Jackpot is a feature of upcoming NetBeans 6.0, so you need to install a recent daily- or Q-build to test it.

Local History

jroller.com/page/ramlog

Update Center URL: *http://people.freenet.de/ramon.ramos/nb/updates.xml*

The Local History module remembers the last changes made to files in your projects, eliminating the need to connect to a version control system (VCS) or performing check-ins. Similar to a persistent undo facility, this module allows you to roll back sources to a previous known good state even after forgetting a commit to the VCS.



Version	Annotation
Sun Apr 23 18:50:00 BRT 2006	
Sun Apr 23 18:50:39 BRT 2006	

2 difference(s)

```
Sun Apr 23 18:50:00 BRT 2006                    Current Version
250    }                                        250    }
251                                             251
252*   public BigDecimal divide (BigDecimal         *
253      throws ArithmeticException, Illegal
254    {
255      return divide (val, scale, rounding
256    }
257
258    public BigDecimal divide(BigDecimal v    252    public BigDecimal divi
259      throws ArithmeticException, Illegal    253      throws ArithmeticExc
260    {                                        254    {
261      if (roundingMode < 0 || roundingMod    255      if (roundingMode < 0
```

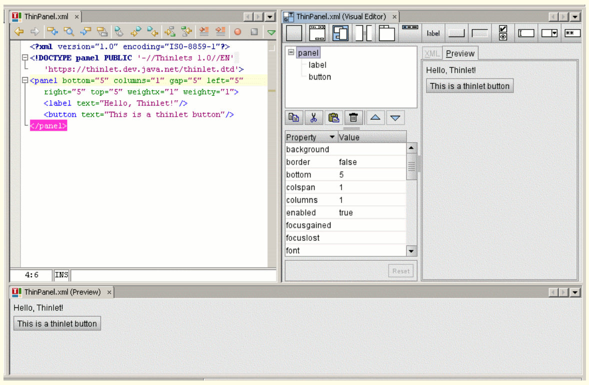
Local History: remembering last changes in code

ThinNB

thinnb.dev.java.net

Update Center Configuration: *https://thinnb.dev.java.net/nonav/net-java-dev-thinnb.nbm*

The ThinNB module provides full support for Thinlet development, including a visual GUI builder. (Thinlet is a lightweight, XML-based GUI framework.)



```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE panel PUBLIC '-//Thinlets 1.0//EN'
  'https://thinlet.dev.java.net/thinlet.dtd'>
<panel bottom="5" columns="1" gap="5" left="5"
  right="5" top="5" weightx="1" weighty="1">
  <label text="Hello, Thinlet!"/>
  <button text="This is a thinlet button"/>
</panel>
```

Property	Value
background	
border	false
bottom	5
colspan	1
columns	1
enabled	true
focusgained	
focuslost	
font	

ThinNB: A complete environment for Thinlet development

NB Extras

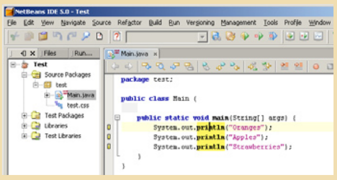
nbextras.org

Update Center Configuration: *http://updatecenter.nbextras.org/5.0/org-netbeans-modules-nbextras-updatecenter.nbm*

NBExtras is a special case. This portal provides a blog and an update center for a collection of plug-in modules that enhance standard NetBeans features. Here is a selection of the modules available.

Mark Occurrences

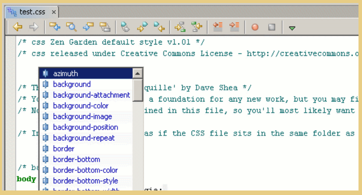
Highlights every occurrence of an identifier selected in a Java editor.



```
package test;

public class Main {

    public static void main(String[] args) {
        System.out.println("Oranges");
        System.out.println("Apples");
        System.out.println("Strawberries");
    }
}
```

JavaScript Editor and CSS code completion

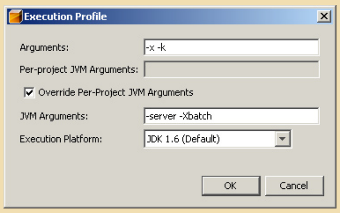
Essential for Web developers, these support syntax-highlighting and code completion.



```
/* css Zen Garden default style v1.01 */
/* css released under Creative Commons License - http://creativecommons.o
```

Execution Profile

Allows configuring different launch options (application and JVM arguments) for your projects' main classes.



Arguments: -x -k

Per-project JVM Arguments:

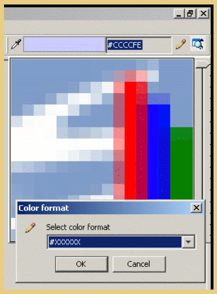
[✓] Override Per-Project JVM Arguments

JVM Arguments: -server -Xbatch

Execution Platform: JDK 1.6 (Default)

Desktop Sampler

A color sampler and screen magnifier tool; useful for GUI developers. The modules work well together, as it's easier to sample colors of individual pixels in the magnified view.

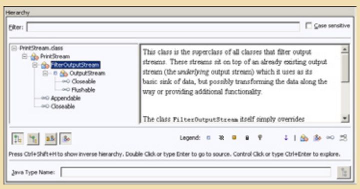


#CCCCFE

Select color format: #XXXXXX

Java Type Hierarchy

Opens a popup with the full hierarchy of classes and interfaces (descendants or ancestors) for a Java class.



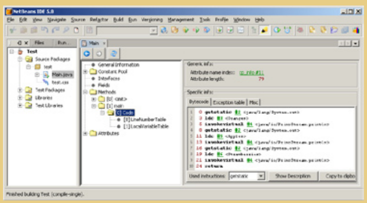
This class is the superclass of all classes that filter output streams. These streams sit on top of an already existing output stream (the *underlying* output stream) which it uses as its basic sink of data, but possibly transforming the data along the way or providing additional functionality.

The class `FilterOutputStream` itself simply overrides

Press Ctrl+Shift+H to show inverse hierarchy. Double Click or type Enter to go to source. Control Click or type Ctrl+Enter to explore.

ByeCode Browser

Allows browsing the contents of *.class* files, to inspect bytecode and class metadata. For real hackers.

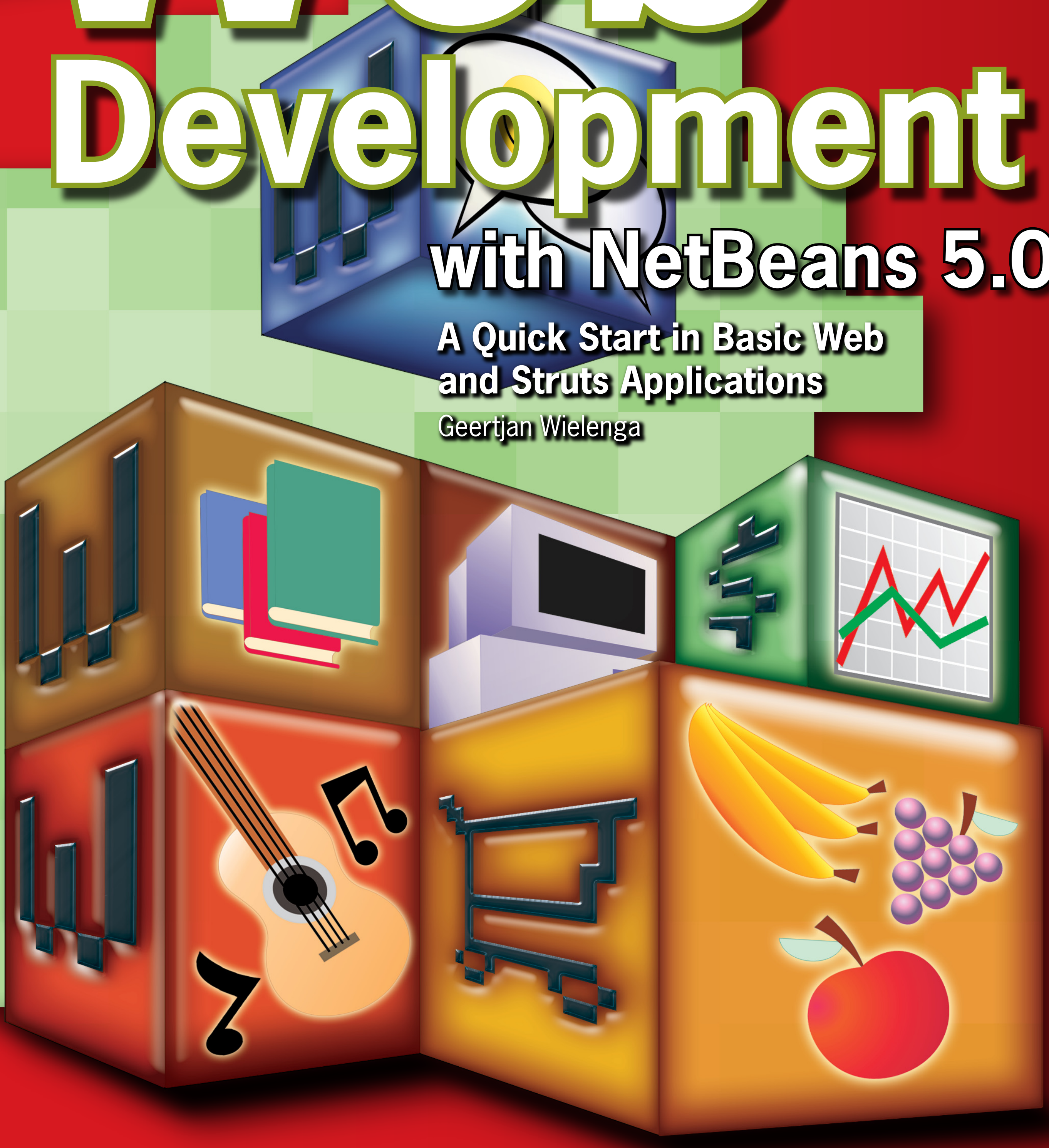


- **JAD decompiler front-end:** Invokes the JAD bytecode decompiler, showing the generated Java sources in the IDE.
- **Line Tools:** Augments the editor with operations like moving a line or group of lines up or down, sorting a group of lines, and removing duplicated lines.
- **Java File Structure:** Opens a popup window detailing the structure of a Java file, with every possible detail (modifiers, javadocs, fully qualified names etc.).
- **Regular Expression Highlighter:** Typing a regular expression in this module's toolbar selects all matches in the editor.
- **CVS Report / SVN Report:** These generate various statistical reports on source code repositories.
- **Keytool / CryptoServices:** Facilities to work with the JDK's security utilities.
- **Launch4J:** Integrates support for Launch4J, a tool that produces native launchers for Java applications.
- **SystemProperties:** Shows the system properties for the running JVM (the one executing NetBeans itself). Useful for module developers.
- **Recently Opened Files:** Makes NetBeans' *File* menu remember the most recently opened files.
- **FreeForm Project Extras:** Adds extra features for FreeForm projects (which use arbitrary structure and Ant build scripts).
- **Code Template Tools:** Makes it easier to work with code templates in editors.
- **Filesystem-based code completion:** When editing string literals, does completion based on the project's classpath. Very useful for working with resources.
- **Jump to next error:** Creates an editor action that jumps to the next compilation error.
- **Google Search toolbar:** Adds a toolbar that triggers a Google web search.
- **Project Packager:** Exports one or more projects to a ZIP file, filtering out non-transferable files like CVS state.
- **Run Terminal:** For command-line fans – opens a system terminal, with the current directory set to the project directory selected in NetBeans.

Web Development with NetBeans 5.0

A Quick Start in Basic Web and Struts Applications

Geertjan Wielenga



This tutorial takes you through the basics of using NetBeans IDE 5.0 to develop web applications. First you create, deploy, and execute a simple web application using a JSP and a JavaBean. Then you learn in detail how to use the best of NetBeans features to create Struts applications.

Setting Up a Web Application Project

Before you begin, you need to have installed NetBeans IDE 5.0 and a JDK version 1.4.2 or 5.0 (or newer). Optionally, you can download the Sun Java System (SJS) Application Server, JBoss, or WebLogic. However, the Tomcat Web Server that is bundled with the IDE provides all the support you need for two-tier web applications such as those described in this tutorial.

Registering the Server with the IDE

The bundled Tomcat Web Server is registered with the IDE automatically. However, before you can deploy to the SJS Application Server, JBoss, or WebLogic, you have to register a local instance with the IDE. If you installed the NetBeans IDE 5.0/SJS Application Server bundle, a local instance of the SJS Application Server is registered automatically. Otherwise, do the following.

Choose *Tools|Server Manager* from the main window. Click *Add Server*, select the server type and give a name to the instance. Click *Next*. Then specify the server information, the location of the local instance of the application server, and the domain to which you want to deploy.

Creating a New Web Application Project

We start by creating a new web application. Choose *File|New Project*. Under *Categories*, select *Web*. Under *Projects*, select *Web Application* and click *Next*.

Under *Project Name*, enter "HelloWeb". Notice that the *Context Path* is "/HelloWeb". Change the Project Location to any directory on your computer. From now on, this directory is referred to as *$PROJECTHOME*.

Select the recommendations to which your source structure will adhere, which is purely a personal preference:

- **Java BluePrints**. For more information, see *java.sun.com/blueprints/code/projectconventions.html*.
- **Jakarta**. See *jakarta.apache.org/tomcat/tomcat-5.0-doc*.

Select the server to which you want to deploy your application. Only servers that are registered with the IDE are listed. Leave the *Set as Main Project* checkbox selected. Click *Finish*.

The IDE creates the *$PROJECTHOME/HelloWeb* project folder. The project folder contains all of your sources and project metadata, such as the project's Ant build script. The HelloWeb project opens in the IDE. You can view its logical structure in the *Projects* window and its file structure in the *Files* window.

Creating and Editing Web Application Source Files

The IDE provides a wide range of tools that can fit any developer's personal style, whether you prefer to code everything by hand or want the IDE to generate large chunks of code for you.

Creating a Java Package and a Java Source File

Expand the *Source Packages* node. Note that this node only contains an empty default package node. Right-click the *Source Packages* node and choose *New>Java Class*. Enter "NameHandler" in the *Class Name* text box and type "org.me.hello" in the *Package* drop-down. Click *Finish*.

In the Source Editor, declare a field by typing the following line directly below the class declaration:

```
String name;
```

netbeans.org/kb

NetBeans tutorials, guides and articles

Add the following line in the **nameHandler()** method:

```
name = null;
```

Generating Getter and Setter Methods

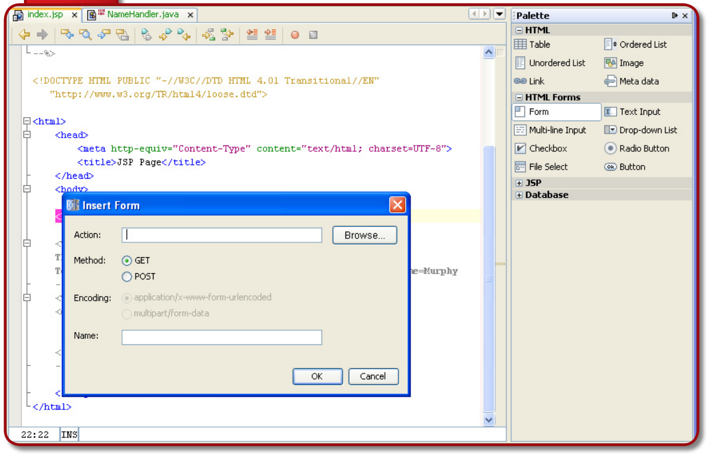
Right-click the word **name** in the field declaration at the start of the class and choose *Refactor>Encapsulate Fields*. Click *Next* to run the command with its default options, and then click *Do Refactoring*. Getter and setter methods are generated for the **name** field and its access level is changed to private. The Java class should now look similar to **Listing 1**.

Editing the Default JavaServer Pages File

Expand the HelloWeb project node and the Web Pages node. Note that the IDE has created a default JavaServer Pages file, *index.jsp*, for you. When you create the project, the IDE opened the *index.jsp* file in the Source Editor.

Listing 1. First version of the *NameHandler* class

```
package org.me.hello;

public class NameHandler {
  private String name;

  /** Creates a new instance of NameHandler */
    public NameHandler() {
      setName(null);
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
}
```



```
--%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
</html>
```

Action:
Method: [✓ GET] POST
Encoding: application/x-www-form-urlencoded / multipart/form-data
Name:

Figure 1 Adding a HTML form to the *index.jsp*

Select the *index.jsp* Source Editor tab. The JSP file now has focus in the Source Editor. In the Palette on the right side of the Source Editor, expand *HTML Forms* and drag a *Form* item below the **<h1>** tags (see **Figure 1**). Set the following values: *Action:* response.jsp; *Method*: GET; *Name*: "Name Input Form". Click *OK*. The Form is added to the *index.jsp* file.

Drag a Text Input item to just before the **</form>** tag. Set the following values: *Name:* name; *Type:* text. Click *OK*. The Text Input is added between the **<form>** tags.

Drag a Button item to just before the **</form>** tag. Set the following values: *Label:* OK; *Type:* submit. Click *OK*. The Button is added between the **<form>** tags.

Type "Enter your name:" in front of the **<input>** tag and change the text between the **<h1>** tags to "Entry Form". The tags between the **<body>** tags now look as follows:

```
<h1>Entry Form</h1>
 <form name="Name Input Form" action="response.jsp"
      method="GET">
 Enter your name:
  <input type="text" name="name" value="" />
  <input type="submit" value="OK" />
</form>
```

Creating a JavaServer Pages File

Expand the HelloWeb project node and the Web Pages node. Right-click the *Web Pages* node and choose *New>JSP*; name the JSP file "*response*", and click *Finish*. The new *response.jsp* opens in the Source Editor.

In the Palette on the right side of the Source Editor, expand *JSP* and drag a *Use Bean* item right below the **<body>** tag in the Source Editor. Set the following values: *ID:* mybean; *Class:* "org.me.hello. NameHandler"; *Scope:* session. Click *OK*.

The Use Bean item is added below the **<body>** tag.

Add a *Get Bean Property* item and a *Set Bean Property* item from the Palette. Then change the code so that the tags between the **<body>** tags look as follows:

```
<jsp:useBean id="mybean" scope="session"
   class="org.me.hello.NameHandler" />
<jsp:setProperty name="mybean" property="*" />
<h1>Hello, <jsp:getProperty name="mybean"
    property="name" />!</h1>
```

Building and Running a Web Application Project

NetBeans uses an Ant build script to build and run your web applications. It generates the build script based on the options you enter in the *New Project* wizard and the project's *Project Properties* dialog box.

Choose *Run>Run Main Project* (F6) from the main menu. The IDE builds the web application and deploys it, using the server you specified when creating the project.

Enter your name in the text box on your deployed *index.jsp* page, as in **Figure 2**. Click *OK*. The *response.jsp* page should open and greet you; see **Figure 3**.

Moving to Struts

The following sections take you through the basics of using NetBeans IDE 5.0 to develop web applications that make use of the Struts framework.

The Struts framework enables you to create maintainable, extensible, and flexible web applications based on standard technologies, such as JSP pages, resource bundles, and XML. Struts works with a Model-View-Controller (MVC) framework, but focuses on the controller – a servlet, which is included in the Struts libraries that the IDE provides and automatically registers in the *web.xml* deployment descriptor when you indicate that you want to use Struts.

The Struts servlet uses the *struts-config.xml* file to map incoming requests to a Struts "action" class. An action class receives a Struts "actionform bean" class as input, which serves as a transfer object between the action class and the view. This is typically a JavaServer Pages (JSP) page. Because many web applications use JSP pages for the view, Struts provides custom tag libraries which facilitate interaction with HTML forms.

Setting Up a Struts Application

In NetBeans, a "Struts application" is nothing more than a normal web application accompanied by the Struts libraries and configuration files. You create a Struts application in the same way you create any other web application in the IDE – with the additional step of indicating that you want the Struts libraries and configuration files to be included in your application.

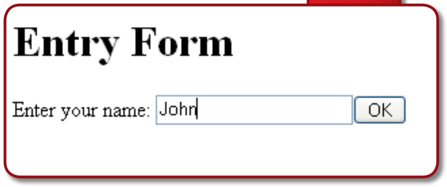
Choose *File|New Project*. Under *Categories* select *Web*. Under *Projects* select *Web Application* and click *Next*. In the Name and Location panel under *Project Name*, enter "LoginPage". Change the *Project Location* to any directory on your computer. As before, this directory will be referred to as *$PROJECTHOME*.

Select the server to which you want to deploy your application. Notice that the Context Path is */LoginPage*. Click *Next*. In the *Frameworks* panel, select Struts 1.2.7 (see **Figure 4**).

Do not change any of the values in the lower section of this panel. They serve the following purposes:

- **Action Servlet Name.** Hardcoded specification of the name of the servlet entry for the Struts action servlet. The *web.xml* deployment descriptor contains a servlet entry for the action servlet, specifying the appropriate Struts specific parameters, such

2



Entry Form

Enter your name: John

Figure 2 Testing the HelloWeb app

3

Hello, John!

Figure 3 Greetings from HelloWeb

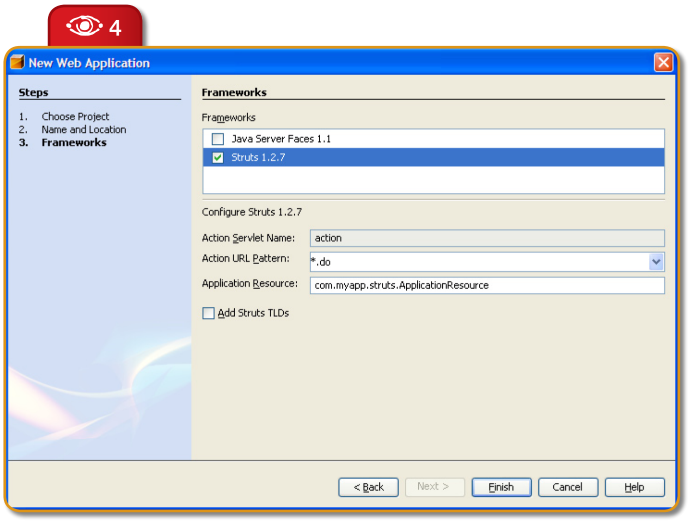


Figure 4 Initial Struts configuration in NetBeans

New Web Application

Steps
1. Choose Project
2. Name and Location
3. **Frameworks**

Frameworks

Frameworks
- [ ] Java Server Faces 1.1
- [✓] Struts 1.2.7

Configure Struts 1.2.7

Action Servlet Name: action
Action URL Pattern: *.do
Application Resource: com.myapp.struts.ApplicationResource
- [ ] Add Struts TLDs

as the name of the servlet class and the path to the *struts-config.xml* configuration file.

▪ **Action URL Pattern.** Allows the appropriate patterns which should be mapped to the Struts action controller to be specified. This generates a corresponding *web.xml* servlet mapping entry to map the specified URI pattern to the action servlet. By default, only the **.do* pattern is mapped.

▪ **Application Resource.** Lets you specify the resource bundle which will be used in the *struts-config.xml* file for localizing messages. By default, this is *com.myapp.struts.ApplicationResource*.

▪ **Add Struts TLDs.** Lets you generate tag library descriptors for the Struts tag libraries. A tag library descriptor is an XML document which contains additional information about the entire tag library as well as each individual tag. In general this is not necessary, because you can refer to URIs rather than local TLD files.

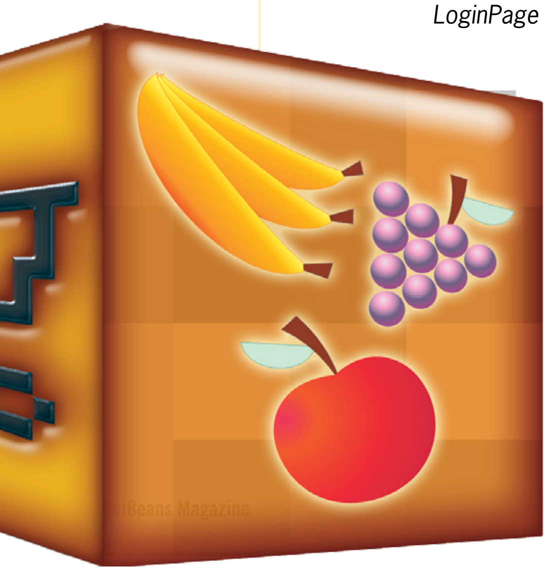
Click *Finish*. The IDE creates the *$PROJECTHOME/LoginPage* project folder in your filesystem. As with any web application in the IDE, the project folder contains all of your sources and the IDE's project metadata, such as the Ant build script. However, in addition, your web application has all of the Struts libraries on its classpath. Not only are they on the application's classpath, but they are also included in the project and will be packaged with it when you build it later.

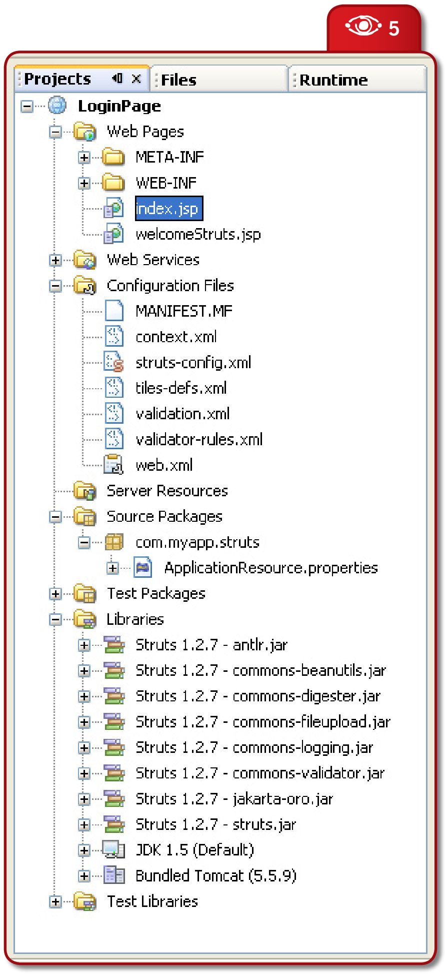
The LoginPage project opens in the IDE. The *Projects* window should now look as in **Figure 5**. In the *Configuration Files* node, the application includes all the Struts-specific configuration files, of which *struts-config.xml* is the most important. You will use this configuration file throughout this tutorial. Also in *Configuration Files*, to handle Struts processing, the Struts servlet (i.e., the controller in the MVC paradigm) is mapped in the *web.xml* deployment descriptor (see **Listing 2**).

Here the Struts servlet (**org.apache.struts.action.ActionServlet**) is specified as the servlet that will control all requests for the mapping *.do*. In addition, the *web.xml* file specifies that the Struts servlet is configured by means of the *struts-config.xml* file that is found in the *WEB-INF* folder.

Listing 2. Struts servlet mapping in web.xml

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
     org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml
    </param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
     <param-name>detail</param-name>
     <param-value>2</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```




5

```
LoginPage
  Web Pages
    META-INF
    WEB-INF
    index.jsp
    welcomeStruts.jsp
  Web Services
  Configuration Files
    MANIFEST.MF
    context.xml
    struts-config.xml
    tiles-defs.xml
    validation.xml
    validator-rules.xml
    web.xml
  Server Resources
  Source Packages
    com.myapp.struts
      ApplicationResource.properties
  Test Packages
  Libraries
    Struts 1.2.7 - antlr.jar
    Struts 1.2.7 - commons-beanutils.jar
    Struts 1.2.7 - commons-digester.jar
    Struts 1.2.7 - commons-fileupload.jar
    Struts 1.2.7 - commons-logging.jar
    Struts 1.2.7 - commons-validator.jar
    Struts 1.2.7 - jakarta-oro.jar
    Struts 1.2.7 - struts.jar
    JDK 1.5 (Default)
    Bundled Tomcat (5.5.9)
  Test Libraries
```

Figure 5 Logical structure of the LoginPage project

Developing a Struts Application

Developing a Struts application is similar to developing any other kind of web application in the IDE. You use components such as JSP pages, servlets, listeners, and filters. However, you complement your web development toolkit by using the facilities provided by Struts via the IDE.

For example, you use templates in the IDE to create Struts action classes and Struts actionform bean classes. On top of that, the IDE automatically registers these classes in the *struts-config.xml* file and lets you extend this file very easily via menu items in the Source Editor's pop-up menus.

Using Struts Custom Tag Libraries in a JSP Page

Many web applications use JSP for the view in MVC, so Struts provides custom tag libraries which facilitate interaction with HTML forms. These can very easily and smoothly be set up and used in a JSP page in the IDE. See how, following these steps:

Right-click the LoginPage project node, choose *New>JSP*, and call the new JSP page "loginForm". Click *Finish*. In the Source Editor, change the default content of the **<h1>** tag to "Login Form" and do the same for the text in the **<title>** tags.

Copy these first two taglib directives from the *welcomeStruts.jsp* file to the top of your new *loginForm.jsp* file:

```
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean" %>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html" %>
```

In *loginForm.jsp*, below the **<h1>** tags, add the following:

```
<html:form action="login">
  <html:submit value="Login" />
</html:form>
```

Notice that when you type the Struts tags, the IDE helps you by suggesting different ways of completing the code, and further helps you by providing Struts Javadoc; see **Figure 6**.

Whenever you finish typing in the Source Editor, you can neaten the code by right-clicking in the Source Editor and choosing *Reformat Code*.

In the Component Palette on the right side of the Source Editor, within the HTML section, drag the *Table* item to just above the **<html:submit value="Login" />** line. The *Insert Table* dialog box pops up. Set both the *Rows* and *Columns* to 1, so that you'll create a table consisting of one row and one column. Click *OK*.

In between the **<th>** tags, type the following:

```
<bean:message key="login.name" />
```

In between the **<td>** tags, type:

```
<html:text property="name" />
```

The body of *loginForm.jsp* is now as shown in **Figure 7**.

In the *Projects* window, expand the *Source Packages* node, expand the *com.myapp.struts* node, and double-click the *ApplicationResource.properties* file so that it opens in the Source Editor. Add "login.name" as a key, anywhere in the

struts.apache.org Struts Framework

ApplicationResource.properties file, and add a meaningful message. For example, "login.name=Name".

Using Struts to Validate a Field in a JSP Page

A Struts "actionform bean" class represents data shared between the view (in this case, a JSP page) and the Struts action class. An actionform bean class is available both for populating the view and for providing input to an action class; it may also implement a **validate()** method to allow input mapped from the view to be verified.

Right-click the LoginPage project node and choose *New>File/Folder*. In the *Web* category choose *Struts ActionForm Bean* and click *Next*. Notice that your actionform bean class will be called "NewStrutsActionForm". Leave the default name for purposes of this tutorial. In a real application, you would give the class a new, meaningful name. Select *com.myapp.struts* in the *Package* drop-down list and click *Finish*.

The class opens in the Source Editor. By default, the IDE provides it with a string called **name** and an **int** called **number**. Both fields also have getters and setters defined for them.

Open *struts-config.xml* in the Source Editor and note that it contains, among other things, the following:

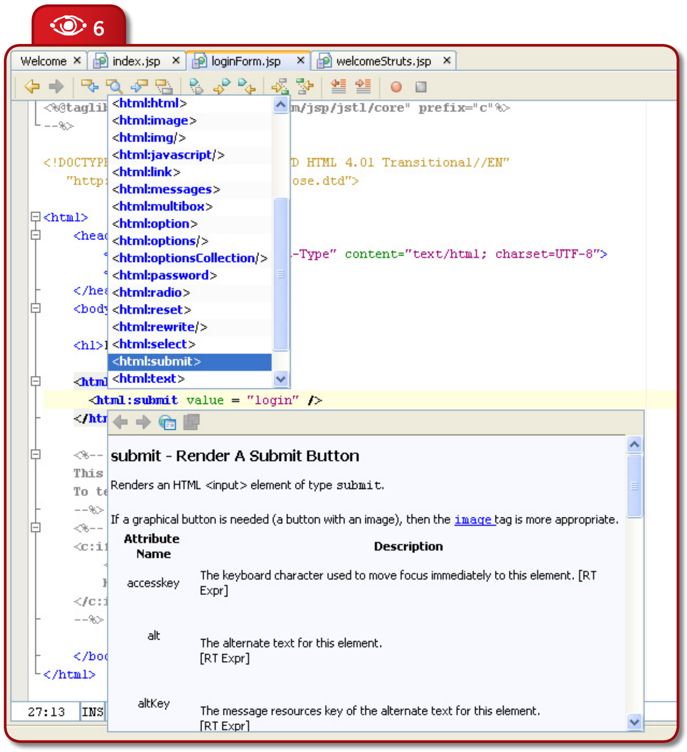
```
<form-beans>
   <form-bean name="NewStrutsActionForm"
        type="com.myapp.struts.NewStrutsActionForm"/>
</form-beans>
```

Hold down the Ctrl key and move your mouse over the fully qualified class name of the actionform bean class. A hyperlink appears. Click it to navigate to the actionform bean class.

Now browse through the actionform bean class in the Source Editor. Look at the **validate()** method that the IDE created for you:

```
public ActionErrors validate(ActionMapping mapping,
     HttpServletRequest request)
{
  ActionErrors errors = new ActionErrors();
  if (getName() == null || getName().length() < 1) {
      errors.add("name",
        new ActionMessage("error.name.required"));
      // TODO: add 'error.name.required'
      // key to your resources
   }
   return errors;
}
```

Notice that the field called **name** is validated by default. If validation fails,



```
<html:submit value = "login" />
```

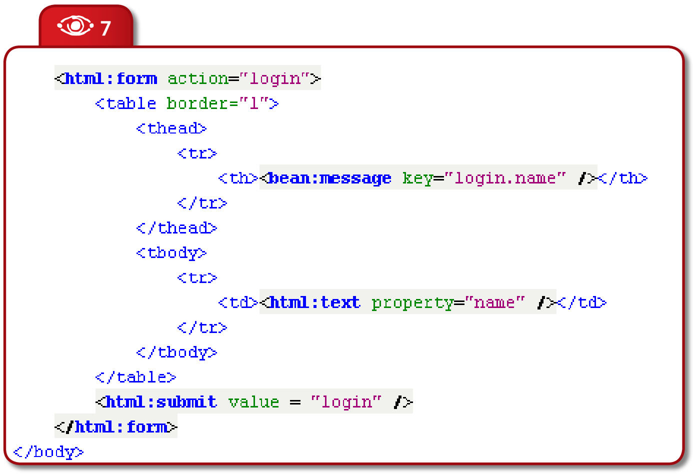
submit - Render A Submit Button

Renders an HTML <input> element of type submit.

If a graphical button is needed (a button with an image), then the image tag is more appropriate.

Attribute Name	Description
accesskey	The keyboard character used to move focus immediately to this element. [RT Expr]
alt	The alternate text for this element. [RT Expr]
altKey	The message resources key of the alternate text for this element. [RT Expr]

Figure 6 The IDE provides code completion and documentation for Struts taglibs



```
<html:form action="login">
    <table border="1">
        <thead>
            <tr>
                <th><bean:message key="login.name" /></th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td><html:text property="name" /></td>
            </tr>
        </tbody>
    </table>
    <html:submit value = "login" />
</html:form>
</body>
```

Figure 7 *LoginForm.jsp* body after first changes

which happens when no name is entered in the JSP page, a message that is identified by **error.name.required** is returned.

Following the **TODO** instruction that the IDE put in the **validate()** method for you, add **error.name.required** as a key to the *ApplicationResource.properties* file with a meaningful message. For example: "error.name.required=Enter a name!".

At the top of the file, to customize the formatting of your error message, change the first four keys to the following:

```
errors.header=
errors.prefix=<span style="color: red">
errors.suffix=</span>
errors.footer=
```

Specify where you want the error message to be rendered, by adding the following line in *loginForm.jsp*, right above the closing **</html:form>** tag:

```
<html:errors />
```

Using Struts to Navigate between JSP Pages

A Struts "action" class is executed in response to a user request and commonly interacts with the model through a business delegate. The responsibility of an action class is to provide navigation and forward control to the appropriate view.

Right-click the LoginPage project node and choose *New>File/Folder*. In the *Web* category choose *Struts Action* and click *Next*. In the *Name and Location* panel, notice that your action class will be called "NewStrutsAction". Leave the default name for purposes of this tutorial.

Select *com.myapp.struts* in the *Package* drop-down list. Type "login" in *Action Path* (the content of *Action Path* is therefore now "/login"). Click *Next*. In the step "ActionForm Bean, Parameter", notice that the IDE suggests that you associate the action class with the actionform created in the previous step. In *Input Resource*, browse to your *loginForm.jsp* page. Click *Finish*.

Open *struts-config.xml* in the Source Editor and note that it contains, among other things, the following:

```
<action-mappings>
  <action input="/loginForm.jsp"
      name="NewStrutsActionForm"
      path="/login"  scope="session"
      type="com.myapp.struts.NewStrutsAction"/>
  <action path="/Welcome"
      forward="/welcomeStruts.jsp"/>
</action-mappings>
```

Supposing you want the action class to function per request, instead of per session, put the cursor in the **scope** attribute and press Ctrl-Space (see **Figure 8**).

Choose *Request*. Hold down the Ctrl key and move your mouse over the action class's fully qualified class name. Click the hyperlink to navigate to the action class.

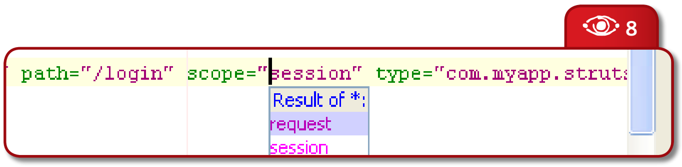
Browse through the action class and look at the **execute()** method:

```
public ActionForward execute(ActionMapping mapping, ActionForm  form,
   HttpServletRequest request, HttpServletResponse response) throws Exception
{
      return mapping.findForward(SUCCESS);
}
```

Notice the definition of **SUCCESS**, at the top of the **NewStrutsAction**:

```
private final static String SUCCESS = "success";
```

This specifies that this action class forwards to the output view called "success". You need to define a page that will be displayed when this output view is called. So, create another JSP in the same location as *loginForm.jsp* and call it "loginSuccessful". In the Source Editor, change the default content of the **<h1>** tags in the new



```
 path="/login" scope="session" type="com.myapp.struts
```

java.sun.com/javaee Java EE Technologies at Sun

Figure 8
Using code completion to change an action class scope

JSP files to "Login Successful!" and do the same for the text in the **<title>** tags.

Open *struts-config.xml* in the Source Editor; right-click anywhere in the */login* action mapping, and choose *Struts>Add Forward* (see **Figure 9**).

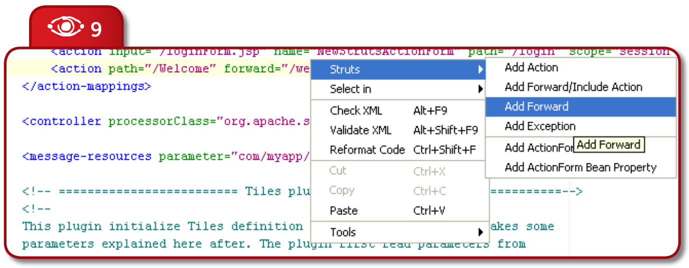
In the *Add Forward* dialog type "success" in *Forward Name*, and browse to *loginSuccessful.jsp* in *Resource File*. The dialog box should now look as in **Figure 10**.

Click *Add*. Notice that *struts-config.xml* now shows the following (the new code is in bold):

```
<action-mappings>
  <action input="/loginForm.jsp"
  ...
      <forward name="success" path="/loginSuccessful.jsp"/>
  </action>
...
</action-mappings>
```

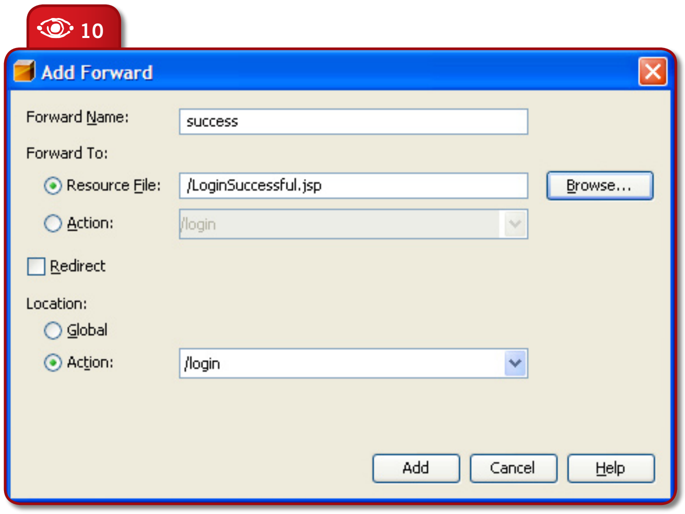
Building and Running the Struts Application

NetBeans uses an Ant build script to build and run your web application. The IDE generated the build script when you created the app, basing it on the options entered in the *New Project* wizard and the project's *Project Properties* dialog box.



```
    <action input="/loginForm.jsp" name="NewStrutsActionForm" path="/login" scope="session
    <action path="/Welcome" forward="/we
</action-mappings>

<controller processorClass="org.apache.s

<message-resources parameter="com/myapp/

<!-- ========================= Tiles plu                    ==========-->
<!--
This plugin initialize Tiles definition                    akes some
parameters explained here after. The plugin first read parameters from
```

Figure 9
Adding a forward in *struts-config.xml*



Forward Name: success

Forward To:

(●) Resource File: /LoginSuccessful.jsp [Browse...]

( ) Action: /login

[ ] Redirect

Location:

( ) Global

(●) Action: /login

Figure 10
Setting forward options

Right-click the project node and choose *Properties*. In the *Project Properties* dialog box, click the *Run* node and in *Relative URL* type *login.do*. Click *OK*. (Remember that you mapped the *.do* mapping to the Struts controller servlet. Now, when you run the application and the *.do* mapping is used, the Struts controller servlet knows it has to handle the request.)

Choose *Run|Run Main Project* (F6) from the main menu. The IDE builds the application and deploys it, using the server you specified when creating the project. The browser opens to display the *loginForm.jsp* page (see **Figure 11**).

Only if field-level validation succeeds, so that the action class's **execute()** method returns the success output view, does Struts call the *loginsuccesful.jsp* page. To pass validation, all that you need to do is add any value to the Name row in the *loginForm.jsp* page. Then, *loginSuccessful.jsp* is displayed.

Of course, this is not a complete login form; it merely shows you what the basis of such a form could look like in Struts. The following section shows you how the form can be extended with a variety of standard functionality.

Adding More Functionality to the Struts Application

Struts simplifies and organizes an application in many more ways than can be listed here. However, here are some simple extensions possible to ouexisting login page, using Struts.

Using Struts to Add "Cancel" Functionality

In *loginForm.jsp*, below the **<html:submit>**

line, create a *Cancel* button by adding the following:

```
<html:cancel />
```

Add these lines to the execute method in **org.myapp.struts.NewStrutsAction**:

```
if (isCancelled(request)){
  return mapping.findForward(CANCEL);
}
```

Press Ctrl-Space within the **isCancelled()** method and then read the Javadoc to understand the method. Declare the definition of **CANCEL** at the top of the **NewStrutsAction** class, right below the definition of **SUCCESS**:

```
private final static String CANCEL = "cancel";
```

You need to define a page that will be displayed when **CANCEL** is called. So, create another JSP in the same location as *loginForm.jsp* and call it **loginCancel()**. In the Source Editor, change the default content of the **<h1>** tags in the new JSP files to "Login Cancelled!", and do the same for the text in the **<title>** tags.

Open *struts-config.xml* in the Source Editor, right-click anywhere in the */login* action mapping, and choose *Struts>Add Forward*. The *Add Forward* dialog box opens. Type "cancel" in *Forward Name*. Browse to *loginCancel.jsp* in *Resource File* and click *Add*. Notice the change in *struts-config.xml*:

```
<action-mappings>
 <action input="/loginForm.jsp"
   ...
    <forward name="success"
       path="/loginSuccessful.jsp"/>
    <forward name="cancel"
         path="/loginCancel.jsp"/>
 </action>
 ...
</action-mappings>
```

Run the application again and notice the new *Cancel* button. Click it and the new *loginCancel.jsp* page is opened in the browser.

Using Struts to Add "Logout" Functionality

In *loginForm.jsp*, below the **<h1>** tags, create the *Logout* link by adding the following:

```
<html:link action="/logout">
  Logout</html:link>
```

You need to define a page that will be displayed when the Logout link is clicked. So, create another JSP in the same location as *loginForm.jsp* and call it "loginOut". In the Source Editor, change the default content of the **<h1>** tags in the new JSP files to "Have a Nice Day!" and do the same for the text in the **<title>** tags.

Open *struts-config.xml* in the Source Editor; right-click anywhere, and choose *Struts>Add Forward/Include Action*. The *Add Forward/Include Action* dialog box opens. Type "logout" in *Action Path* and browse to *loginOut.jsp* in *Resource File*. Then click *Add*. Notice the changes in *struts-config.xml:*

```
<action-mappings> <action input="/loginForm.jsp"
  ...
 </action> <action forward="/loginOut.jsp"  path="/logout"/>
...
</action-mappings>
```

Run the application again and notice the new Logout link (see **Figure 12**). Click it, and the new *loginOut.jsp* page is opened in the browser.

Summary

In this tutorial you learned to create a simple webapp. You also saw, step by step, how to set up and develop a Struts application in NetBeans and how to build and run it, using the most of the IDE's features.

11

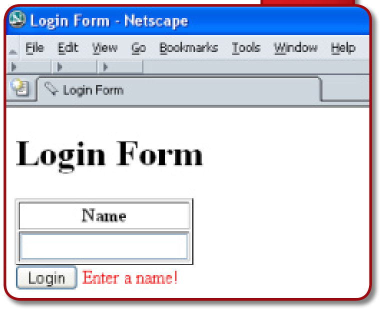


Figure 11 Login form after entering an empty name

12

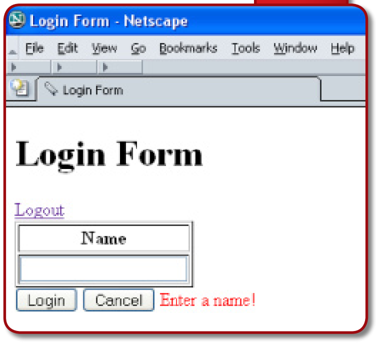


Figure 12 New Login Form, with cancel and logout functionality

Geertjan Wielenga (*geertjan.wielenga@sun.com*) has been a software technical writer for the past 10 years, working mainly on rapid application development and IDEs, currently on NetBeans, but previously on UNIFACE by Compuware and SuperNova by the now-defunct Four Seasons Software. He works for Sun Microsystems as a Technical Writer on the NetBeans project.

Writing Quality Code with NetBeans

Learn about tools and best practices that help increase the quality of your code

Osvaldo Pinali Doederlein

Modern IDEs like NetBeans have great source code editors, debuggers, profilers, visual builders and other tools that help producing application that are complete, correct and efficient. But this is not enough: source code should also be well organized and well structured, easy to read and maintain, and compliant with a myriad of "best practices" that help to deliver these qualities and avoid problems. In other words: your code should be above suspicion.

Unit tests and code reviews help make sure your code is well written. But you can also get help from several tools that check code formatting, detect bugs, improve OO style, automate tests – and generally relieve you from at least part of the manual work of code quality assurance.

In this article we look at several tools supported by NetBeans, either out of the box (JUnit, refactoring and the NetBeans Profiler) or through plug-in modules (Checkstyle, PMD and FindBugs). We can only provide an introduction to each tool, but the take-home message is that you can improve significantly your Java software construction process without a huge investment in effort.

Learning to use these tools well – configuring, combining and matching their capabilities under the integrated platform of NetBeans – will let you add important items to your programming discipline without large costs in additional development time. In fact, the proper use of good tools has the potential to reduce total effort, by saving you from hunting difficult bugs, or by making your source code better structured, and easier to read and evolve.

Defensive Programming

When I started programming in the late 80's, any coding mistake would result in a cryptic error code or a system crash. But few veterans should miss compilers from the eighties as much as they do for Atari 2600 games. Modern compilers like *javac* produce clear diagnostics for dozens of errors that are statically verifiable. In Java SE 5, new language features (remarkably generic types) expand the number of errors that can be caught at compile time, and this is a Good Thing.

However, we can always use more of a good thing. Even though runtime technologies like exception handling and a safe memory model in the JVM, or ACID transactions in a DBMS, handle runtime errors gracefully and prevent disastrous results, the right time to "catch" any bug is development time. Techniques like unit testing help, but the more static diagnostics, the better.

There are of course tools that go beyond *javac*, and attempt to detect *code anti-patterns*: snippets of code that despite not being forbidden by the Java Language Specification (JLS), are fingerprints of bad code. Here "bad" might mean buggy, slow, unstructured, or just difficult to understand and maintain. PMD and FindBugs are perhaps the most popular of the open source offerings, and NetBeans plug-in modules make their use a breeze in the IDE: you will hardly notice that you are reviewing code for dozens of potential programming issues.

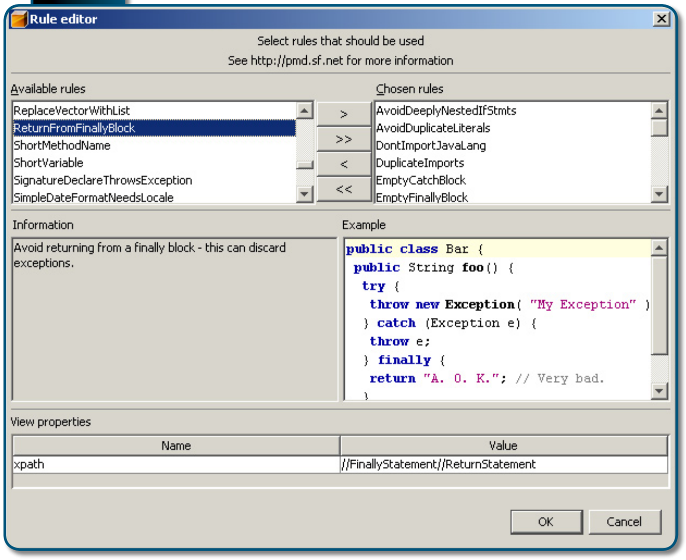
Using PMD

We will start by looking at PMD. Its NetBeans plug-in can be fetched by the Update Center (from *www.nbextras.org*). Once installed, **Figure 1** shows the configuration dialog for the PMD plug-in module, and one of PMD's over 180 rules. After everything is set up (to get started, you can use the plug-in's installation defaults), select a project's *Source Packages* folder and run *Tools|Run PMD*.

java.sun.com/docs/codeconv

Code conventions from Sun.

Figure 1
PMD Configuration in NetBeans



Rule editor

Select rules that should be used
See http://pmd.sf.net for more information

Available rules
- ReplaceVectorWithList
- ReturnFromFinallyBlock
- ShortMethodName
- ShortVariable
- SignatureDeclareThrowsException
- SimpleDateFormatNeedsLocale

Chosen rules
- AvoidDeeplyNestedIfStmts
- AvoidDuplicateLiterals
- DontImportJavaLang
- DuplicateImports
- EmptyCatchBlock
- EmptyFinallyBlock

Information

Avoid returning from a finally block - this can discard exceptions.

Example

```
public class Bar {
 public String foo() {
  try {
   throw new Exception( "My Exception" );
  } catch (Exception e) {
   throw e;
  } finally {
   return "A. O. K."; // Very bad.
  }
```

View properties

Name	Value
xpath	//FinallyStatement//ReturnStatement

OK Cancel

The *ReturnFromFinallyBlock* rule illustrated in the figure is a good example of the kind of diagnostics PMD performs: it says it's a bad thing to have a **return** statement inside a **finally** block. The Example pane illustrates the problem, and the Information pane explains it. Such **return**s shortcut any exception thrown or leaked by the **try**/**catch** structure, discarding exceptions that were supposed to be delivered to the caller (or by an outer **try**/**catch**). This is a bug pattern, because it's not a very intuitive or useful programming style. In fact many programmers will ignore that the Java language will behave that way. In addition, exceptions are of course supposed to be handled by a **catch** block – so we can condemn as bad style even the rare code written with the "**return** inside **finally**" idiom on purpose.

Validating the validator

Not all PMD rules are so crystal-clear to pick, however. Some are even documented as controversial. For example, the *UnusedModifier* rule flags redundant modifiers, such as a **public** method in an **interface** (since all interface members are **public**). I like this rule; it makes declarations simpler and not redundant. But others may prefer their source code to be as explicit as possible.

There are other reasons to disable rules, like false positives in some rules with less than perfect detection, or rules that are relevant for some projects but not for others. A good example of both is the rule *NonThreadSafeSingleton*, which finds Singletons with non-**synchronized** getter/initializer methods. This rule may mistake some regular methods for a Singleton getter, or be irrelevant for single-threaded apps*[1]. However, most of the time this rule will catch real bugs, and when it does so, you will be very glad to have enabled it.

In short, you have to decide which rules should be enforced or not. This is a difficult balance. Having more rules activated increases the chances that PMD catches important problems in your code, with no effort from you. But too many rules may produce a smorgasbord of warnings that require too much work to review, ultimately making people stop using the tool.

The code-review process must be light enough so you can do it often and quickly. You shouldn't let one week of construction pass without re-checking all the new-or-updated code (if not the entire project), at least for the most important smoking guns.

Here's a tip to help adoption of PMD (and similar tools). Start with a "pet project" whose source code is of high quality and small volume. Then activate all of PMD's rules, run it on the project's sources, and check the resulting warnings. You will have a clear idea of which rules are inadequate for your personal programming habits. Disable these rules, and run PMD with the same configuration on a larger project – one written by many developers with different skills, with lots of legacy and hacked bug fixes – and check if the number of warnings is reasonable enough so you can start enforcing those rules in the project.

*[1] Not that it makes much sense to write *any* non-thread-safe code today, with multi-core CPUs becoming commodity.

Remember that this is mostly a one-time effort for legacy projects. Once all programmers are disciplined to run PMD regularly, and code is written since Revision-1 with a chosen ruleset in mind, the project tends to keep a small or empty set of warnings as it grows. The reviewing process consequently becomes very easy and incremental.

Showing off PMD

Figure 2 shows the NetBeans source editor with a Java class analyzed by PMD. This class was never checked before and all rules are active, so the PMD Output window shows quite a large number of warnings. This output includes some warnings many wouldn't agree with and disable (excessively *long* variable names?!), as well as one that is hugely important, and which is further commented in the **sidebar** "The final modifier and refactoring: towards functional-style Java".

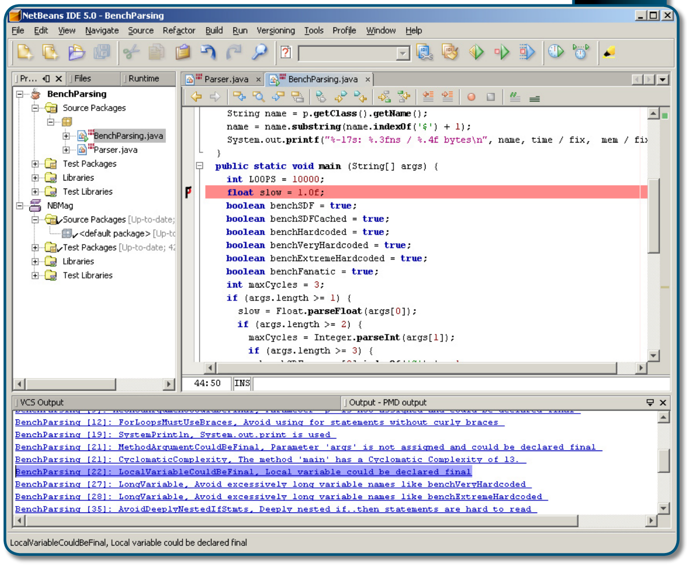
Here is a very small sample of interesting rules supported by PMD:

- **SwitchDensity:** Finds **switch** statements that have too much code per **case**, in average. You should consider refactoring large **case** blocks into separate methods.
- **InsufficientStringBufferDeclaration:** Even if you build complex strings with **StringBuffer** (or **StringBuilder** in JSE 5), your code is not optimal if it uses these classes' default constructors, which initialize the buffer with a small size, forcing reallocation as data is appended. This rule even recommends a minimum initial size for the buffer, looking at the sizes of all literal strings appended by your code.
- **ForLoopShouldBeWhileLoop:** Detects **for** loops that could be rewritten as simpler **while** loops.
- **PositionLiteralsFirstInComparisons**: Prefer **"literal".equals(str)** to **str.equals("literal")**, because the former will never throw a **NullPointerException**.
- **AbstractClassWithoutAbstractMethod:** Often indicative of weak OO design. An abstract class with no methods should rather be an **interface**. And if it contains only concrete methods, perhaps it should not be **abstract**. Great abstract classes are those that implement design patterns like GoF's Template Method, with a combination of abstract methods and concrete methods that depend on subclasses implementing the former.
- **UnsynchronizedStaticDateFormatter:** A **SimpleDateFormat** object that is often used with the same format cannot be initialized once, be stored in a **static** variable, and then be reused by multiple method invocations that use the same format. The problem is that this API is not thread-safe, so concurrent invocations will break it.

Using FindBugs

FindBugs does essentially the same as PMD, so why use two very similar tools? Because each tool has different strengths. PMD's scope is broader and it is easier to extend. You can create your own rules with relatively little effort (see the sidebar mentioned above), which is also the reason that PMD supports more rules (twice as many as FindBugs) out of the box.

2



```
    String name = p.getClass().getName();
    name = name.substring(name.indexOf('$') + 1);
    System.out.printf("%-17s: %.3fns / %.4f bytes\n", name, time / fix,  mem / fi
  }
  public static void main (String[] args) {
    int LOOPS = 10000;
    float slow = 1.0f;
    boolean benchSDF = true;
    boolean benchSDFCached = true;
    boolean benchHardcoded = true;
    boolean benchVeryHardcoded = true;
    boolean benchExtremeHardcoded = true;
    boolean benchFanatic = true;
    int maxCycles = 3;
    if (args.length >= 1) {
      slow = Float.parseFloat(args[0]);
      if (args.length >= 2) {
        maxCycles = Integer.parseInt(args[1]);
        if (args.length >= 3) {
```

```
BenchParsing [12]: ForLoopsMustUseBraces, Avoid using for statements without curly braces
BenchParsing [19]: SystemPrintln, System.out.print is used
BenchParsing [21]: MethodArgumentCouldBeFinal, Parameter 'args' is not assigned and could be declared final
BenchParsing [21]: CyclomaticComplexity, The method 'main' has a Cyclomatic Complexity of 13.
BenchParsing [22]: LocalVariableCouldBeFinal, Local variable could be declared final
BenchParsing [27]: LongVariable, Avoid excessively long variable names like benchVeryHardcoded
BenchParsing [28]: LongVariable, Avoid excessively long variable names like benchExtremeHardcoded
BenchParsing [35]: AvoidDeeplyNestedIfStmts, Deeply nested if..then statements are hard to read
```

LocalVariableCouldBeFinal, Local variable could be declared final

www.nbextras.org Aggregates dozens of NetBeans plug-ins, including PMD and FindBugs.

Figure 2 PMD in action, warning about violations of the selected rules.

On the other hand, FindBugs has a more advanced architecture, enabling more sophisticated and precise detections. Instead of pattern matching, its "detectors" are implemented on top of bytecode scanning and dataflow analysis. This enables FindBugs to locate such problems as infinite loops, as well as many subtle null-pointer bugs, and even security issues like JDBC code allowing SQL injection attacks. All that with a very small number of false positives. (As a trade-off, you would face a steep learning curve to create detectors for your own rules.)

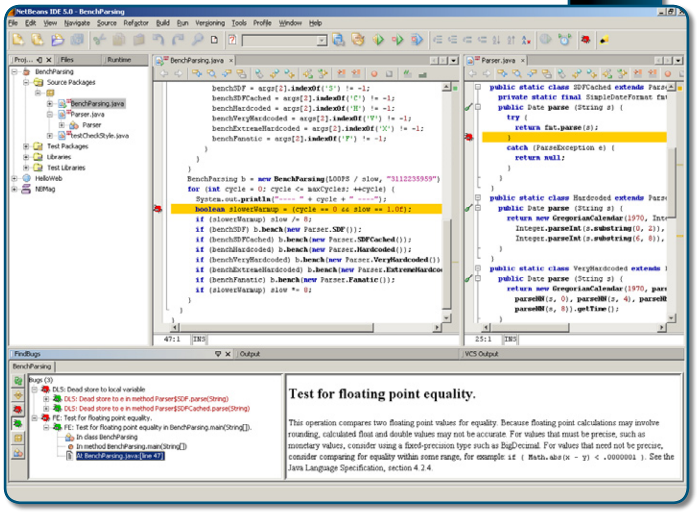
Showing off FindBugs

FindBugs' plug-in can be downloaded from the same Update Manager site as PMD. **Figure 3** shows that FindBugs' plug-in for NetBeans is prettier, with a custom view that helps to review warnings. The selected warning is for the bug rule "FE", or "Test for floating point equality". In Java (and any language using floating-point types), you should avoid comparing floating point values (**float**, **double**) with [in]equality operators (**==** and **!=**), like explained by the command pane of the plug-in.

In another example, FindBugs sees two violations of DLS (Dead Local Store): redundant assignments to variables that are never read. The flagged variables are exception arguments of **catch** blocks, so FindBugs actually found that I'm ignoring these exceptions silently, which is of course a bad practice (at least the exception should be logged somewhere).

In **Figure 4** we can see FindBugs' settings. In addition to a full list of individual rules in the "Configure Detectors" page, you can set up FindBugs more easily with the Feature-oriented page. In my experience, setting the Level option to Medium (excluding only the rules with Experimental status and Low severity) is enough to keep the number of violations small enough to start using the tool, even in large projects that were never before massaged by a code validation tool.

3


NetBeans IDE 5.0 - BenchParsing
File Edit View Navigate Source Refactor Build Run Versioning Tools Profile Window Help
BenchParsing.java
Parser.java
FindBugs
Output
VCS Output
BenchParsing

```
        benchSDF = args[2].indexOf('S') != -1;
        benchSDFCached = args[2].indexOf('C') != -1;
        benchHardcoded = args[2].indexOf('H') != -1;
        benchVeryHardcoded = args[2].indexOf('V') != -1;
        benchExtremeHardcoded = args[2].indexOf('X') != -1;
        benchFanatic = args[2].indexOf('F') != -1;
      }
    }
  }
  BenchParsing b = new BenchParsing(LOOPS / slow, "3112235959")
  for (int cycle = 0; cycle <= maxCycles; ++cycle) {
    System.out.println("---- " + cycle + " ----");
    boolean slowerWarmup = (cycle == 0 && slow == 1.0f);
    if (slowerWarmup) slow /= 8;
    if (benchSDF) b.bench(new Parser.SDF());
    if (benchSDFCached) b.bench(new Parser.SDFCached());
    if (benchHardcoded) b.bench(new Parser.Hardcoded());
    if (benchVeryHardcoded) b.bench(new Parser.VeryHardcoded())
    if (benchExtremeHardcoded) b.bench(new Parser.ExtremeHardco
    if (benchFanatic) b.bench(new Parser.Fanatic());
    if (slowerWarmup) slow *= 8;
  }
 }
}
47:1 INS
```

```
public static class SDFCached extends Pars
  private static final SimpleDateFormat fmt
  public Date parse (String s) {
    try {
      return fmt.parse(s);
    }
    catch (ParseException e) {
      return null;
    }
  }
}
public static class Hardcoded extends Pars
  public Date parse (String s) {
    return new GregorianCalendar(1970, Inte
      Integer.parseInt(s.substring(0, 2)),
      Integer.parseInt(s.substring(6, 8)),
  }
}
public static class VeryHardcoded extends
  public Date parse (String s) {
    return new GregorianCalendar(1970, pars
      parseNN(s, 0), parseNN(s, 4), parseN
      parseNN(s, 8)).getTime();
  }
25:1 INS
```

```
Bugs (3)
  DLS: Dead store to local variable
    DLS: Dead store to e in method Parser$SDF.parse(String)
    DLS: Dead store to e in method Parser$SDFCached.parse(String)
  FE: Test for floating point equality.
    FE: Test for floating point equality in BenchParsing.main(String[]).
      In class BenchParsing
      In method BenchParsing.main(String[])
      At BenchParsing.java:[line 47]
```

Test for floating point equality.

This operation compares two floating point values for equality. Because floating point calculations may involve rounding, calculated float and double values may not be accurate. For values that must be precise, such as monetary values, consider using a fixed-precision type such as BigDecimal. For values that need not be precise, consider comparing for equality within some range, for example: `if ( Math.abs(x - y) < .0000001 )`. See the Java Language Specification, section 4.2.4.

Figure 3 FindBugs in action, identifying embarrassing code from the author.

Here is a list of particularly valuable detectors of FindBugs:

- **BC (Impossible Cast)**

Finds code that if executed will always fail with a **ClassCastException**.

- **BIT (Incompatible bit masks)**

Finds code like "**if (A & B == C)**" that will always result false, because either B and C, or A and C, have no **1** bits in common.

- **DMI: Code contains a hard coded reference to an absolute pathname**

This goes in the category "*I did that while prototyping/debugging, but forgot to clean up later…*".

- **EC: Invocation of equals() on an array, which is equivalent to ==**

Unfortunately, the Java platform does not define proper behavior of **equals()** for primitive arrays (like **int[]**), but programmers often forget this.

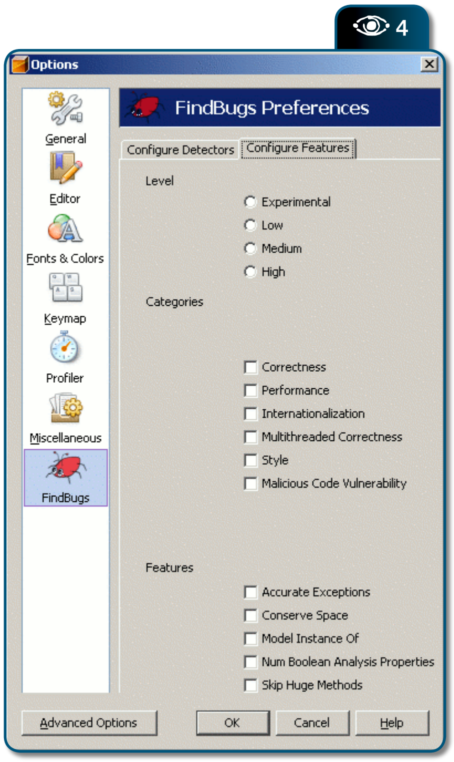
- **IC: Initialization circularity**

This is a very insidious bug, when class A has static initialization code that depends on class B, and B's initialization likewise depends on A's initialization.

A real-world case study

- **NP: Load of known null value**

This is a very useful detector, and it just found an important but elusive bug in the project I'm currently working on. Look at this real-world code:



Level

- ◯ Experimental
- ◯ Low
- ◯ Medium
- ◯ High

Categories

- ☐ Correctness
- ☐ Performance
- ☐ Internationalization
- ☐ Multithreaded Correctness
- ☐ Style
- ☐ Malicious Code Vulnerability

Features

- ☐ Accurate Exceptions
- ☐ Conserve Space
- ☐ Model Instance Of
- ☐ Num Boolean Analysis Properties
- ☐ Skip Huge Methods

Figure 4 Easy configuration of FindBugs' rules.

```
private Map headers = new HashMap();
public void putHeader (String id, String value) {
  if (value == null)
    headers.remove(value);
  else
    headers.put(id, value);
}
```

Where is the null-pointer bug? This code will never throw **NullPointerException**; the problem is more subtle. My intention was that **putHeader(*id*, null)** would remove the header **id**; but I mistyped the code as **headers.remove(value)** – it should have been **headers.remove(id)**.

FindBugs finds this bug because it doesn't make sense to use a variable whose content is known to be **null**, *in lieu* of the literal **null** value. If I really wanted to remove the **null** value from my **HashMap**[2], I should instead write **headers.remove(null)**.

A bug like this is very hard to find because nothing looks wrong on a cursory examination. No exception will ever be thrown; and no obvious functional problem will happen (in my app, the only effect of keeping unwanted data in this **Map** would be a brief memory leak). What is even more vicious is that the method looks so simple – being a thin wrapper over a **Map** – that I didn't bother to write a unit test for it.

Fortunately, tools like FindBugs or PMD don't fail in discipline. They will review your code without missing even the "too easy to need testing" code.

Programming with Style

Besides fixing bugs, most programmers also care about keeping their source code well indented and formatted, making it easier to read and modify. Opinions differ about many style rules, but most people will agree that *any* style is better than no style.

Checkstyle can check all sorts of style rules, like indentation, spacing, naming conventions, modifier usage, curly braces and so on. It can also find violations of several programming best practices and potential bugs, so there's some intersection of functionality with PMD and FindBugs, which can also check some style-related problems. In general, Checkstyle is much better on style rules, and the latter tools are more powerful otherwise. My recommendation is to enable all tools, but configure their respective configurations to not have overlap (otherwise your reviewing effort will increase with the duplicated warnings).

Checkstyle's NetBeans plug-in must be downloaded from the project site, and installed from the local NBM file. The plug-in doesn't currently offer a GUI for choosing rules. You must first configure a XML file that specifies style rules, like these[3]:

```
<!— Formatting rule: Open '{' in a new line -->
<module name="LeftCurly"/>
  <property name="option" value="nl"/>
</module>
```

You don't actually have to write such a file, as Checkstyle comes with standard configurations, the most popular being *sun_checks.xml* (conventions from Sun Microsystems). You can start with such a default configuration and tweak a few rules for a perfect match with your preferences (detailed documentation for each rule is provided by Checkstyle's docs).

Figure 5 shows the output from Checkstyle for a test class that purposefully violates many of my personal style rules: no javadocs for public methods and no *package.xml* file; **equals()** without

[2] Remember that this collection supports a **null** key.

[3] PMD and FindBugs vwork the same way, with XML configuration files, although their GUIs hide these. But even with the GUI plug-ins you may want to access the XML files, for example to add the configuration to your version control system, to copy it to other projects, or to reuse them in Ant or Maven scripts.

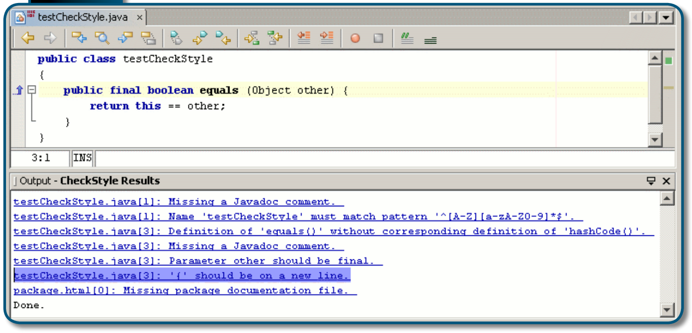
hashCode(); non-**final** parameter; class name not compliant with Java naming convention (starting with lowercase); and finally, opening a curly brace the Wrong Way (in *my* book, anyway)!

Any good developer is able to keep her own source code neatly formatted, but in large projects with many coders, often including people with different experiences, contractors etc., it's easy to loose control and let the codebase become a mess. A tool like Checkstyle is remarkably important and effective to enforce a minimal standard of code style.

Fearless Programming

Not every coding problem can be detected by static analysis tools, so you still have to plan, implement, and execute runtime tests. Fortunately, there are other tools that automate many testing tasks, and NetBeans helps you further by integrating these in a seamless environment.

In this section, we'll review NetBeans' integration with JUnit and explore how your unit testing efforts can be reused for fundamental performance testing.


5

```
testCheckStyle.java

public class testCheckStyle
{
    public final boolean equals (Object other) {
        return this == other;
    }
}

3:1  INS
```

Output - CheckStyle Results

```
testCheckStyle.java[1]: Missing a Javadoc comment.
testCheckStyle.java[1]: Name 'testCheckStyle' must match pattern '^[A-Z][a-zA-Z0-9]*$'.
testCheckStyle.java[3]: Definition of 'equals()' without corresponding definition of 'hashCode()'.
testCheckStyle.java[3]: Missing a Javadoc comment.
testCheckStyle.java[3]: Parameter other should be final.
testCheckStyle.java[3]: '{' should be on a new line.
package.html[0]: Missing package documentation file.
Done.
```

Figure 5 Results of executing Checkstyle from inside NetBeans

Listing 1. A class with sample functionality to be tested.

```
public static class HardcodedParser extends Parser {
  public Date parse (String s) {
    return new GregorianCalendar(1970,
      Integer.parseInt(s.substring(2, 4)) - 1,
      Integer.parseInt(s.substring(0, 2)),
      Integer.parseInt(s.substring(4, 6)),
      Integer.parseInt(s.substring(6, 8)),
      Integer.parseInt(s.substring(8, 10))).getTime();
    }
  }
}
```

Unit testing

Unit tests (of course) are runtime tests that cover fine-grained code artifacts: classes and individual methods. These tests are sometimes classified as "white-box tests", because writing them requires intimate knowledge of the code: you should invoke all important methods directly, and know which parameters are required and what results are expected for every call.

In the Java platform, JUnit is the *de facto* unit testing framework, and most if its success should be attributed to simplicity: writing test cases require using a minimalist API and following very simple conventions, remarkably that a test case is a method with signature like "**public void test*Name* ()**". But the integrated JUnit support in NetBeans makes it even simpler for beginners.

Suppose you have written a class like that shown in **Listing 1**. This class is part of a library of several implementations of a **Date**-parsing interface (for a fixed input format), with successive optimization refinements* [4]. Now, optimization is a risky business, so I will sleep better if these methods are tested. With NetBeans, you can select the class and call *Tools>Create JUnit tests*. This wizard will create a class (in the project's Test Packages) with several methods like the one shown in **Listing 2**.

You must only fill in the blanks providing some real input and output data, like in **Listing 3** (where I also trimmed the code a little). Then you only have to run the unit tests with *Run>Test "project name"*, and check the outcome in the JUnit Test Results view (**Figure 6**). Writing and performing tests was never easier!

*[4] I wrote this code for another article, focused on code optimization. The full code is not relevant to the present discussion, but it is available electronically for the curious reader.

Unit Profiling

Notice that JUnit reports not only whether each test has passed, but also its execution time. This suggests we can reuse unit tests for performance testing. Unfortunately, a single execution of a simple method like my **parse()**s may be too fast for the precision of JUnit's timer (see the "0,0s" result for most tests in **Figure 6**). Not to mention other difficulties with Java microbenchmarks, like the need of warm-up time to let dynamic JIT compilers work. But you can fix this by instrumenting the test code, adding loops to run long enough to allow performance measurement, like in **Listing 4**.

Notice that because we're repeating each tested method 100.000 times, the result timing printed by JUnit should be interpreted as hundredths of milliseconds. Also you should run your performance tests with the same JVM options that are expected to be used in production; server-side programs will typically use at least the *-server* option to enable the HotSpot Server JVM (edit this in the project's *Properties> Run>VM options*).

You can see the result of this performance test in **Figure 7**. Now the difference in performance between the various parsing algorithms is very clear (the SDF algorithm is the standard invocation of **SimpleDateFormat.parse()**; the others are increasingly optimized).

Being able to reuse JUnit tests for performance benchmarking is cool: you don't have to write timing code with **System.currentTimeMillis()** (or **nanoTime()** in JSE 5) before and after runs; neither print formatted

Listing 2. Skeleton unit test, generated by NetBeans.

```
public class HardcodedParserTest extends TestCase {
  /**
   * Test of parse method, of class HardcodedParser.
   */
  public void testParse() {
    System.out.println("parse");
    String s = "";

    HardcodedParser instance = new HardcodedParser();
    Date expResult = null;
    Date result = instance.parse(s);
    assertEquals(expResult, result);

    // TODO review the generated test code and
    // remove the default call to fail.
    fail("The test case is a prototype.");
  }
}
```

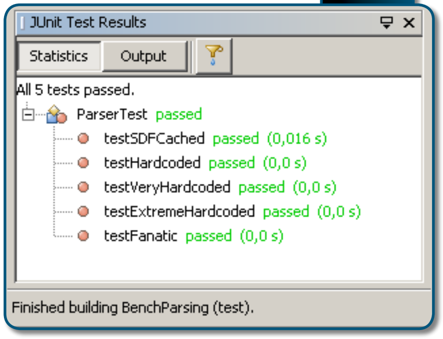
Listing 3. Complete unit test.

```
public class HardcodedParserTest extends TestCase {
  public void testHardcoded() {
    Date expResult = new SimpleDateFormat(
      "ddMMHHmmss").parse("3112235959");

    Date result = new HardcodedParser().
      parse("3112235959");
    assertEquals(expResult, result);
  }
}
```

Listing 4. Unit tests, instrumented for performance measurement.

```
public class ParserPerfTest extends TestCase {
  private static final int LOOPS = 100000;
  public void testHardcoded() {
    HardcodedParser instance = new HardcodedParser();
    Date expResult = new SimpleDateFormat(
      "ddMMHHmmss").parse("3112235959");
    Date result;

    for (int = 0; i < LOOPS; ++i)
      result = instance.parse("3112235959");
    assertEquals(expResult, result);
  }
}
```

6



```
JUnit Test Results
Statistics  Output
All 5 tests passed.
ParserTest passed
    testSDFCached passed (0,016 s)
    testHardcoded passed (0,0 s)
    testVeryHardcoded passed (0,0 s)
    testExtremeHardcoded passed (0,0 s)
    testFanatic passed (0,0 s)
Finished building BenchParsing (test).
```

Figure 6 Unit test results.

The final modifier and refactoring: towards functional-style Java

Of all PMD rules, some of my top favorites are **LocalVariableCouldBeFinal**, **MethodArgumentCouldBeFinal** and **ImmutableField**. The rules suggest declaring local variables, parameters and fields as **final** whenever possible. Many programmers follow this recommendation solely for fields that are initialized only by constructors. The **final** tag prevents bugs where a method would inadvertently update the value of a field that should be constant over an object's lifeycle, such as a person's date of birth.

But what's the deal with **final** parameters and locals? Check this code:

```
// Before:
Employee lookupByPhone (String phone) {
  // Normalizes the phone number
  phone = removeNonDigits(phone);
  Employee employee = findEmployeByPhone(phoneNorm);
  logger.debug("lookupByPhone(" + phone + ") = " + employee);
  return employee;
}

// After:
Employee lookupByPhone (final String phone) {
  final String phoneNorm = removeNonDigits(phone);
  final Employee employee = findEmployeByPhone(phoneNorm);
  logger.debug("lookupByPhone(" + phone + ") = " + employee);
  return employee;
}
```

The method **lookupByPhone()** shows the value of **final** modifiers. First off, they explicitly segregate three semantically distinct entities, which happen to be implemented by a single "local variable" construct in the Java language: Parameters, Local Variables, and Local Constants.

Using **final** where appropriate delivers three main benefits:

1. Avoids misusing a parameter as a local variable (e.g., as a for loop counter).

This is confusing, especially for debugging: if you break into the middle of a method's execution, you won't see the original value of parameters that are later assigned to. Even clicking back in the call stack may not reveal parameter values easily, because these may originate from expressions that weren't stored in variables by the caller (e.g., **f(g() + h())**). With final parameters, the inputs for a method or constructor are always preserved.

2. Shows clearly which identifiers are constants (names for fixed values or shared sub-expressions), and which are real variables (data whose values *vary* with time).

The ability to see constants is important because the fixed binding of a name/value pair is an invariant that you can rely on when analyzing the behavior of complex code.

3. Results in clearer, self-documenting code.

In the example, the original code was modifying the **phone** parameter; to make this parameter **final**, we had to introduce a new local variable, **phoneNorm**. The good thing is that we can encode the meaning of this change in the new variable's name: the phone number without non-digit characters is a *normalized* phone number (i.e., compatible with PKs in the database, keys in **Map**s, etc.). But we don't have to write a comment like "*Normalizes the phone number*", because the new identifier conveys this information – not only in its declaration, but anywhere else it appears.

Notice that **phoneNorm** is also declared **final**, because it's the single transformation we have to do to the phone data. Indeed, most of the time we can replace a variable by multiple constants. This in turn leads to a "functional-style" of Java programming, one that uses as few destructive assignments* as possible.

A second example illustrates better the benefits of such a programming style:

```
final Employee employeeToEvaluate =
  company.detDepartment(depID).getEmployees().get(empName);
if (employeeToEvaluate.calcProductivity() < 7.5)
  company.fire(employeeToEvaluate);
```

Notice that the **employee** variable is redundant: it exists with the sole purpose of avoiding the duplication of a long expression. This variable is really a simple alias to a navigation path in our object graph (**company** → **department[depID]** → **employees** (*a* *Map*) → **get(empName)**). Coding this expression into a **final** variable, i.e. a constant, implicitly documents this fact.

Another interesting aspect of **final** variables is their interaction with refactorings. If you have duplicated expressions in a method, you can use NetBeans' *Introduce Variable* refactoring to remove this duplication. But **Figure B1** shows the right way to use this refactoring: checking the option "*Declare Final*". It's there for a good reason.

If you buy my idea of functional-style Java, there are other tricks that allow even more variables to be eliminated or replaced by **final**s. For example, what to do with a piece of code that uses multiple assignments to a variable, due to different paths of execution?

```
double raise = 1.0; // default
if (employee instanceof Manager)
  raise = 1.1;
else if (employee instanceof CEO)
  raise = 1.25;
 else if (employee instanceof JavaProgrammer)
  raise = 1.6;
employe.setSalary(employee.getSalary() * raise);
```

The answer is easy: use NetBean's *Extract Method* refactoring to move the **if**/then/**else** structure to a new method, say **calcRaise(Employee)**. (*A good OO programmer will further promote this to a polymorphic method of* ***Employee****, overridden by each subtype, instead of using* ***instanceof*** *tests*.) The resulting code is much simpler:

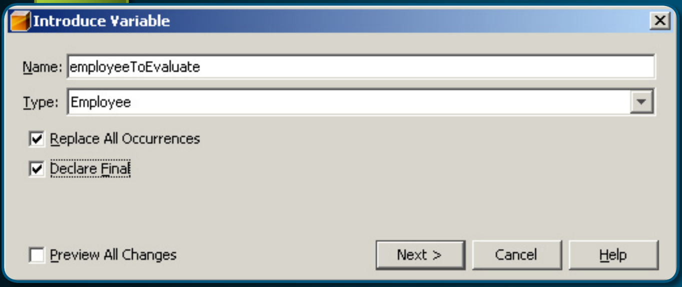
```
employee.setSalary(employee.getSalary() * calcRaise(employee));
...
double calcRaise (Employee emp) {
  if (employee instanceof Manager)
    return 1.1;
  else if (employee instanceof CEO)
    return 1.25;
  else if (employee instanceof JavaProgrammer)
    return 1.6;
  else
    return 1.0; // default
}
```

Notice that the *Extract Method* refactoring will not do the transformation of assignments to the **raise** variable to **return** statements; you have to do that yourself to obtain the code above. We also invoked this new method directly from the **employee.setSalary(…)** expression instead of using a **final** variable to hold that temporary value. We can not only make a variable **final**, but eliminate it completely! Notice we don't loose any code clarity, because the same semantic information encoded in the name of the **raise** variable is now encoded in the name of the **calcRaise()** method. Notice also that **calcRaise()** is a "purely functional" method: it contains no destructive assignments and no side effects, and exhibits the monotonic property (if invoked repeatedly with the same input values, it will deliver the same results, and not affect anything else in the system).

These properties are very interesting for a number of reasons. Now, I won't sell you the full Functional Programming paradigm (I don't practice it either), as it includes harder deals, like forcing the replacement of all iteration by recursion. My point is just snatching into Java an easy part of that paradigm that delivers many of the benefits.

* Assignments that overwrite a previous value, bound to the same identifier by a past assignment.

B1


Introduce Variable

Name: employeeToEvaluate

Type: Employee

☑ Replace All Occurrences

☑ Declare Final

☐ Preview All Changes

Figure B1
Refactoring duplicated expressions? Use final variables.

The transformation we just demonstrated shows more interesting benefits:

▪ **Code is more modular.**

Having lots of small methods is usually better than a few big ones. Even if the new methods are one-trick ponies that don't lead to any reuse, they help making your code self-documenting. Picking good identifiers for these methods and their parameters will embody information that is not always present in equivalent code inside a monolithic method.

Of course, these "extracted methods" will usually be **private**. A class with a large number of **public** methods is ugly, but the number of **private** methods is not important. If the class as a whole is too big, the excess complexity is independent of its internal structure. And the fix for that complexity is breaking the class into more pieces (e.g. via inheritance or delegation) – *not* faking less complexity by packing code into fewer, tighter methods.

▪ **Unit testing is much easier.**

It's hard to argue with this: simpler methods that do less things are much easier to test. In the new version of our code, we can write a set of JUnit tests that target **calcRaise()*** and exercise all possibilities of its algorithm: employees of multiple types and states, including the **null** parameter. It's much harder to do that if your interface with the code is just a higher-level method that sets the raises (the **employee** may not be an immediate parameter of that method). Indeed, the refactored code allows you to *black-box test* a class (i.e. test it knowing only a minimal interface). You don't have to dig inside the source code of each method to analyze every possible execution path and laboriously derive the input data that will force execution of each path.

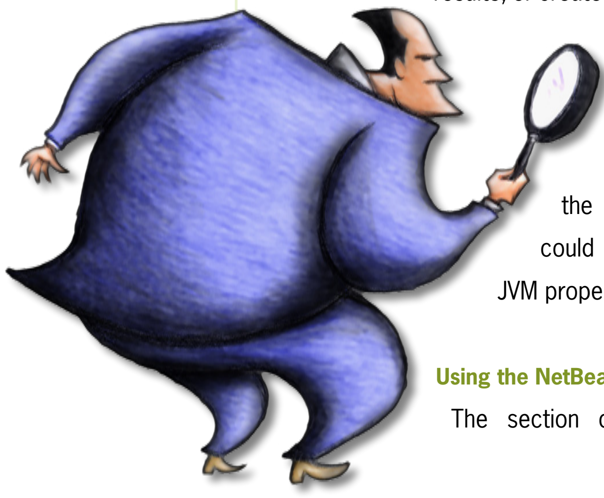
*Not possible if **calcRaise()** is a private method, so standard practice for unit-testers is using package-private instead. But even if you don't want to do relax your member access, the decomposition of code into more methods makes it much easier to analyze it and plan tests that, by entering your class only through public methods, will cover all important code.

Conclusions

The purpose of this discussion was to show that tools like PMD and refactoring are powerful enough even to help enforce sophisticated programming idioms and paradigms. Even if you don't like the concept of "functional-style Java", you may have other ideas of recommended practices, and perhaps rely on these tools to realize such ideas.

It's worth notice that PMD is easy to extend. Many kinds of rules can be programmed in XML ruleset files, with XPath expressions – often one-liners – that match Java abstract syntax tree nodes. Not trivial but much easier than getting acquainted with the complex API of a tool like FindBugs.

Another goal of this discussion was revealing that many recommended rules are worth more than their face value. PMD's documentation for the *LocalVariableCouldBeFinal* rule is a mere "*A local variable assigned only once can be declared final*". But this definitely doesn't capture its potentially profound implications.

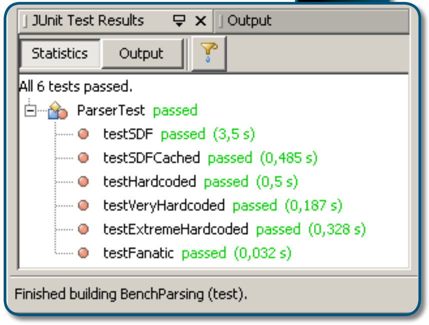


results, or create **main()** methods that invoke all tests of a "suite", not to mention other features of JUnit. When you're not interested in testing performance but only in correctness, just set the **LOOPS** control variable to **1** (this could be done more dynamically, by a JVM property).

Using the NetBeans Profiler

The section on performance testing couldn't end here of course, because we are using NetBeans, which includes a very powerful Profiler. This feature requires an independent installation, but it's a "standard extension" (see details about the tool in the article "Exploring the NetBeans Profiler" in this edition)

Having the Profiler installed, don't start it with the standard option *Profiler>Profile Main project* – because this will run the application's main class. Instead, follow



```
All 6 tests passed.
ParserTest passed
    testSDF passed (3,5 s)
    testSDFCached passed (0,485 s)
    testHardcoded passed (0,5 s)
    testVeryHardcoded passed (0,187 s)
    testExtremeHardcoded passed (0,328 s)
    testFanatic passed (0,032 s)
Finished building BenchParsing (test).
```

these steps:

- Change the **LOOPS** counter to a smaller value like 100, because execution is slower under the profiler (if you don't filter out any code).
- Select the unit test class and execute "Profile file". Select *Analyze performance > Entire application*, and execute it.
- Wait a few seconds until the process terminates.

Figure 8 shows the results of a simple profiling session. We can see the performance of each tested method, and also the breakdown of their executions into invoked APIs and subroutines. The HotSpots view shows a global ranking of the higher-cost methods, and the NetBeans Profiler offers many other tools to analyze and visualize the performance of your code (see the article NetBeans Profiler, in this edition, for additional details and features).

This combination of unit testing and profiling saves effort and also gives you freedom to program incrementally, experimenting new ideas, performing fixes, reviews and refactoring, all without fear of breaking some code or making it slower.

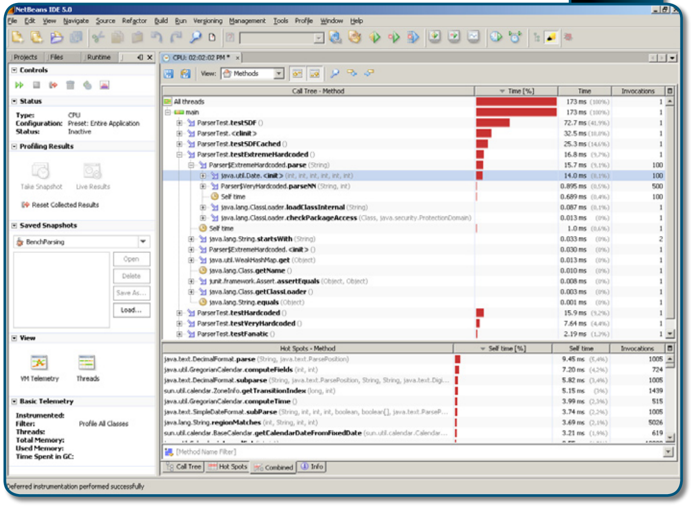
Additional Tools

Other facilities of NetBeans are important for writing good code, and keep its quality as projects grow and evolve. Refactoring and code templates can not only save you typing time, but spare you from bugs caused by inane editing tasks – like fixing all calls to a method with a default value to a new parameter, or typing the thousandth "**for (int i = …)**" loop header in a project. In the runtime and building side, NetBeans' integrated support for Ant is unparalleled, allowing further automation of tasks like compilation, packaging and deployment. (There is a plug-in for Maven, too: Mevenide).

A Version Control System is another must for any self-respecting project, even single-developer ones. NetBeans supports several VCSs, including CVS and Subversion. But don't miss important extras, like the Local History and CVS/SVN Report (see the special section Plug-in Showcase in this magazine).

Conclusions

The NetBeans IDE contains many powerful, high-level features that go beyond basic editing and debugging capabilities, and even beyond eye-catching GUI tools like wizards and visual builders. Built-in tools include support for refactoring, unit testing and profiling. Extra tools include FindBugs, PMD, Checkstyle and many others, all integrated into the open, extensible architecture of the IDE. It's a real sin to miss all the productivity and quality benefits of adding these tools to your daily routine. It will save much more time in the long run than the few days you'll need to invest in learning and customizing the tools, even those that provide many dozens of complex validation rules.



```
Type:           CPU
Configuration:  Preset: Entire Application
Status:         Inactive

Instrumented:   Profile All Classes
Filter:
Threads:
Total Memory:
Used Memory:
Time Spent in GC:
```

```
Call Tree - Method                                                        Time [%]   Time               Invocations
All threads                                                                          173 ms   (100%)     1
main                                                                                 173 ms   (100%)     1
  ParserTest.testSDF ()                                                              72.7 ms  (41,9%)    1
  ParserTest.<clinit>                                                                32.5 ms  (18,8%)    1
  ParserTest.testSDFCached ()                                                        25.3 ms  (14,6%)    1
  ParserTest.testExtremeHardcoded ()                                                 16.8 ms  (9,7%)     1
    Parser$ExtremeHardcoded.parse (String)                                           15.7 ms  (9,1%)     100
      java.util.Date.<init> (int, int, int, int, int, int)                           14.0 ms  (8,1%)     100
      Parser$VeryHardcoded.parseNN (String, int)                                     0.895 ms (0,5%)     500
      Self time                                                                      0.689 ms (0,4%)     100
      java.lang.ClassLoader.loadClassInternal (String)                               0.087 ms (0,1%)     1
      java.lang.ClassLoader.checkPackageAccess (Class, java.security.ProtectionDomain) 0.013 ms (0%)     1
    Self time                                                                        1.0 ms   (0,6%)     1
    java.lang.String.startsWith (String)                                             0.033 ms (0%)       2
    Parser$ExtremeHardcoded.<init> ()                                                0.030 ms (0%)       1
    java.util.WeakHashMap.get (Object)                                               0.013 ms (0%)       1
    java.lang.Class.getName ()                                                       0.010 ms (0%)       1
    junit.framework.Assert.assertEquals (Object, Object)                             0.008 ms (0%)       1
    java.lang.Class.getClassLoader ()                                                0.003 ms (0%)       1
    java.lang.String.equals (Object)                                                 0.001 ms (0%)       1
  ParserTest.testHardcoded ()                                                        15.9 ms  (9,2%)     1
  ParserTest.testVeryHardcoded ()                                                    7.64 ms  (4,4%)     1
  ParserTest.testFanatic ()                                                          2.19 ms  (1,3%)     1

Hot Spots - Method                                                                   Self time [%]   Self time   Invocations
java.text.DecimalFormat.parse (String, java.text.ParsePosition)                              9.45 ms (5,4%)   1005
java.util.GregorianCalendar.computeFields (int, int)                                         7.20 ms (4,2%)   724
java.text.DecimalFormat.subparse (String, java.text.ParsePosition, String, String, java.text.Digi...   5.82 ms (3,4%)   1005
sun.util.calendar.ZoneInfo.getTransitionIndex (long, int)                                    5.15 ms (3%)     1439
java.util.GregorianCalendar.computeTime ()                                                   3.99 ms (2,3%)   515
java.text.SimpleDateFormat.subParse (String, int, int, int, boolean, boolean[], java.text.ParseP...   3.74 ms (2,2%)   1005
java.lang.String.regionMatches (int, String, int, int)                                       3.69 ms (2,1%)   5026
sun.util.calendar.BaseCalendar.getCalendarDateFromFixedDate (sun.util.calendar.Calendar...   3.21 ms (1,9%)   619

Deferred instrumentation performed successfully
```

Figure 7
Using JUnit to measure performance.

Osvaldo Pinali Doederlein
(*opinali@gmail.com*)
is a software architect and consultant, working with Java since 1.0beta. He is an independent expert for the JCP, having participated in the JSR-175 (JSE 5) effort. He is also a contributing editor for the Brazilian Java Magazine.

Figure 8
Running the JUnit performance benchmark under NetBeans Profiler.

Exploring the NetBeans Profiler

From Installation to a Practical Profiling Example*

Gregg Sporar*

NetBeans Profiler is an optional feature of the NetBeans IDE. It is a powerful tool that provides important information about the runtime behavior of an application. Imposing relatively little overhead, the NetBeans Profiler tracks thread state, CPU performance, and memory usage. It uses innovative technology to allow you to tightly control exactly which parts of an application are profiled, resulting in reduced overhead and easier to interpret results. The profiled application can run locally or on a remote system. And by being tightly integrated into the IDE workflow the NetBeans Profiler makes it easy to identify performance problems and memory leaks.

Installing the NetBeans Profiler

The NetBeans Profiler can profile applications running on the following JVMs:

- A customized JFluid VM, based on the standard JDK 1.4.2 VM. This customized JVM is available as a separate NetBeans module download.
- A standard JDK 5.0_04 VM or newer.
- A standard JDK 6 (also known as Mustang) Early Access VM starting from build 26.

Here are the steps for installation:

1. Download the profiler pack installer for your platform from the NetBeans Profiler homepage.

2. If you have a previous version of

* Reprinted with permission from *JavaPerformanceTuning.com*

+ This article also incorporates material from the NetBeans Profiler help text.

the NetBeans Profiler installed, uninstall it first – see the instructions on the download page.

3. Launch the downloaded installer.
4. Proceed through the steps of the installer wizard to install the module.
5. (Re)Start the NetBeans IDE.

Once installed, the module adds the Profile menu item to the menu bar. The Profile menu allows you to start profiling and work with the results. The module also adds two toolbar buttons that are shortcuts to the *Profile* command ( ) and the *Attach and Profile* command ( ).

Getting Started

The NetBeans Profiler provides a number of internal settings that let you tune profiling to your needs. For example, you may decrease the profiling overhead at the cost of some reduction in the amount of generated information. However, it may take some time to understand the meaning and use of the numerous settings available.

For most applications, certain default settings are sufficient. For this reason, the NetBeans Profiler offers two major profiling options. You can start profiling by choosing a simple predefined profiling task, which has most of the settings preset to optimal values and therefore requires little or no tuning. Alternatively, you can create your own custom profiling configuration, where you are free to modify any of the available settings.

To start profiling an application:

1. Open your project in the NetBeans IDE.
2. Right-click your project's entry in the *Projects* window and then select *Set Main Project*.
3. Choose *Profile|Profile Main Project* from the main menu.
4. Choose a profiling command from the list in the *Select Profiling Task* dialog box.
5. Click *Run*.

When you click *Run*, the target application launches and the selected profiling command starts. The NetBeans Profiler Control Panel opens in the IDE.

To see the results of the profiling command, click the *Live Results* button ( ) in the Profiler Control Panel. This opens the *Profiling Results* tab. To stop the profiling command, choose *Profile|Stop* from the menu or click the *Stop* button ( ). If you start the application with the NetBeans Profiler, when you stop the profiling command the application also stops.

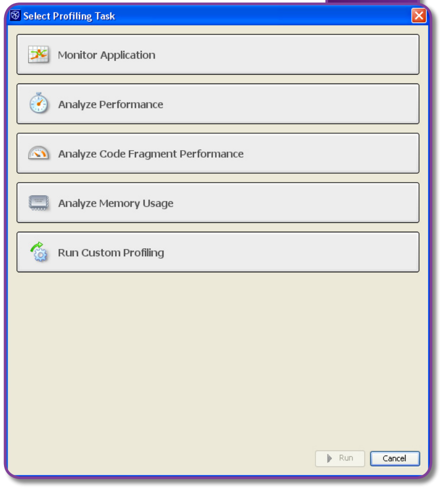
Profiling Tasks

The *Select Profiling Task* dialog box (**Figure 1**) is the main interface for selecting and running profiles. The dialog box gives you five different ways of profiling the target application. The first four are predefined profiling commands. The last command, *Run Custom Profiling*, allows you to create your own custom profiling configuration.

When you click on a profiling command the box expands, displaying a brief explanation. For some tasks it also allows you to set some profiling options. Clicking *Run* at the bottom of the dialog box launches the target application and starts the selected profiling command.

Only one task, i.e. one kind of profiling, can be active for the profiled application at any given time (monitoring is always active however, even when another task is chosen, since its overhead is very low). Note that while profiling you can switch between

1



netbeans.org/products/profiler

NetBeans Profiler

Figure 1
The Select Profiling Task dialog is the main interface for selecting and running profiles

profiling tasks without stopping and restarting your application.

You can choose from the following profiling tasks:

Monitor Application

This command displays high-level information about several important properties of the target JVM, including thread activity and memory usage.

Analyze Performance

Profiles method-level CPU performance (execution time). You can choose to profile the entire application or a part of the application. Detailed filters can be set to control exactly which methods get profiled, allowing the rest of your application to run at full speed.

- **Entire Application:** In this mode, the NetBeans Profiler instruments all of the methods of the profiled application. Threads emit the "method entry" event when entering a method and generate the corresponding "method exit" event when exiting the method. Both of these events contain timestamps. This data is processed in real time.
- **Part of Application:** In this mode, you can instrument and profile a limited subset of the application's code. When partial application profiling is used, profiling data is not collected until one of the application's threads enters a user-selected *root method*. Profiling a limited subset of the application's code may greatly reduce the profiling overhead. Furthermore, for some programs this option may be the *only* way to obtain any detailed and/or realistic performance data at all – because the amount of generated data when the entire application is profiled can be so high as to render the application unusable or even cause it to crash (for example, due to unexpected timeouts).

Application Startup

Use this mode when you want to analyze application startup time.

Analyze Code Fragment Performance

This command measures the time it takes to execute an arbitrary piece of code within one method, or the whole method. By analyzing a code fragment rather than the entire application, the profiling overhead is greatly reduced. The absolute results that you obtain in this mode are closest to the real runtime performance, since the instrumentation is the most lightweight.

Analyze Memory Usage

When you analyze memory usage, the profiling results displayed depend on which of the following options you choose:

- **Record both object creation and garbage collection (Object Liveness):** This option provides information about how many objects of each type are still alive, as well as data on live objects.
- **Record object creation (Object Allocation):** Gives you information about the number, type, and location of objects that have been allocated. This profiling mode is a functional subset of object liveness profiling. The reason for having both modes is that pure object allocation profiling has a smaller performance and memory overhead.

By default only ten percent of all objects for each class are tracked by the NetBeans Profiler. This statistical approach has been shown to deliver results that are as accurate as when all objects are tracked, but with the benefit of greatly reduced profiling overhead.

Run Custom Profiling

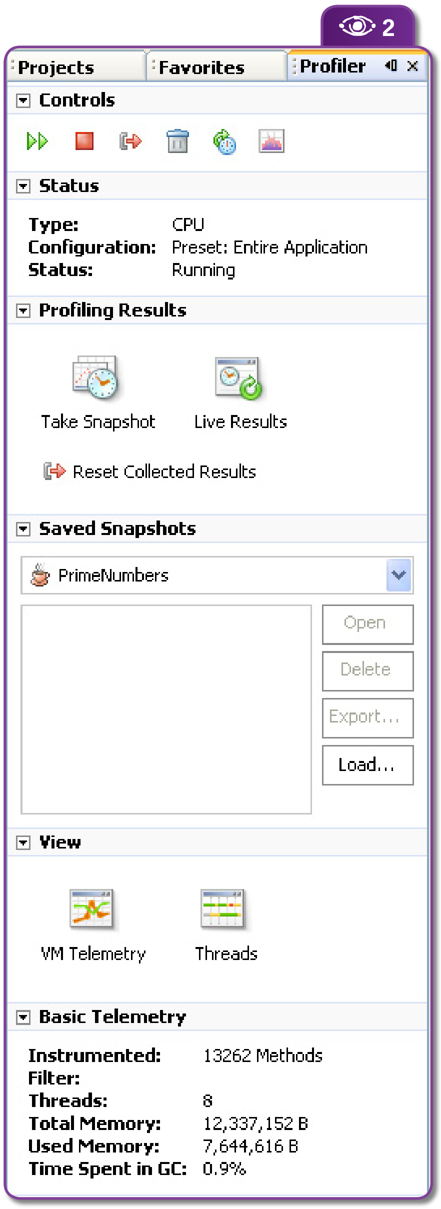
This command is for running custom profiling configurations. You can create, edit, and save these custom configurations, allowing you to control all the internal profile settings.

Control Panel

The profiling control panel is displayed in the left pane of the IDE when you run the NetBeans Profiler (see **Figure 2**). You can open the control panel by choosing *Window |Profiling>Profiler Control Panel*. It contains controls that do the following:

Articles and news about Java performance tools and techniques

javaperformancetuning.com



Projects | Favorites | Profiler

Controls

Status

Type: CPU
Configuration: Preset: Entire Application
Status: Running

Profiling Results

Take Snapshot
Live Results
Reset Collected Results

Saved Snapshots

PrimeNumbers

Open
Delete
Export...
Load...

View

VM Telemetry
Threads

Basic Telemetry

Instrumented: 13262 Methods
Filter:
Threads: 8
Total Memory: 12,337,152 B
Used Memory: 7,644,616 B
Time Spent in GC: 0.9%

Figure 2
The NetBeans Profiler Control Panel

- Control the profiling task
- Display the status of the current profiling task
- Display profiling results
- Manage profiling results snapshots
- Display basic telemetry statistics

Displays

The NetBeans Profiler provides several displays of information. The VM Telemetry Overview is always displayed when the *Monitor Application* command is chosen. To display it at any other time, select *Profile|View>Telemetry Overview*.

See an example in **Figure 3**. In the graph on the left the red shading indicates the allocated size of the JVM heap. The purple overlay indicates the amount of heap space actually in use. In the example the allocated heap size at the last update was over 300 Mb. Of that about 20 Mb is actually being used to hold Java objects.

- The graph on the right shows the count of active threads in the JVM.
- The graph in the center shows two important heap statistics.
- The blue line is the percentage of execution time spent by the JVM doing garbage collection and is graphed against the y-axis on the right edge of the graph. Time spent by the JVM doing garbage collection is time that is not available for it to run your application. So if the blue line indicates a large percentage you may want to consider tuning the JVM by configuring a larger heap size (refer to the *-Xmx* parameter documentation) or perhaps switching to a different garbage collection algorithm.
- The red line is surviving generations and is graphed against the y-axis scale on the left edge of the graph. The count of surviving generations is the number of different ages of all the Java objects on the JVM's heap, where "age" is defined as the number of garbage collections that an object has survived. When the value for surviving generations is low it indicates that most of the objects on the heap have been around about the same amount of time. If, however, the value for surviving generations is increasing at a high rate over time then it indicates your application is allocating new objects while maintaining references to many of the older objects it already allocated. If those older objects are in fact no longer needed then your application is wasting (or "leaking") memory.

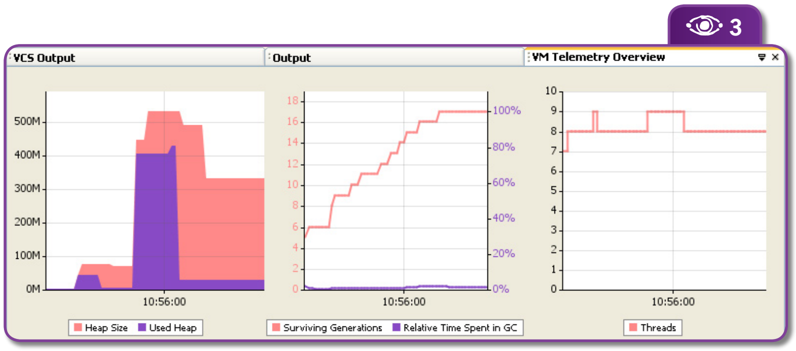


Figure 3
VM Telemetry Overview

Thread State

Thread state is optionally displayed when the Monitor Application command is chosen. It contains the following tabs:

- **Threads (Timeline):** Shows current and historical thread state, updated as the application runs.
- **Threads (Details):** Shows a summary of thread state information for a single thread.

A sample timeline graph is shown in **Figure 4**. Color coding is used to display thread state:

- **Green:** the thread is either running or is ready to run.
- **Purple:** the thread is sleeping; for example it called **Thread.sleep()**.
- **Yellow:** the thread is waiting in a call to **Object.wait()**.
- **Red:** the thread is blocked while trying to enter a synchronized block or method.

Live Results

Clicking the *Live Results* button ( ) in the control panel will open the *Profiling Results* tab. Depending on the profiling command that is running, this tab will display either performance or object allocation and liveness statistics.

The NetBeans Profiler will update the displayed profiling results automatically at short intervals (about 2 seconds) if the *Update Results Automatically* button ( ) in the toolbar is clicked.

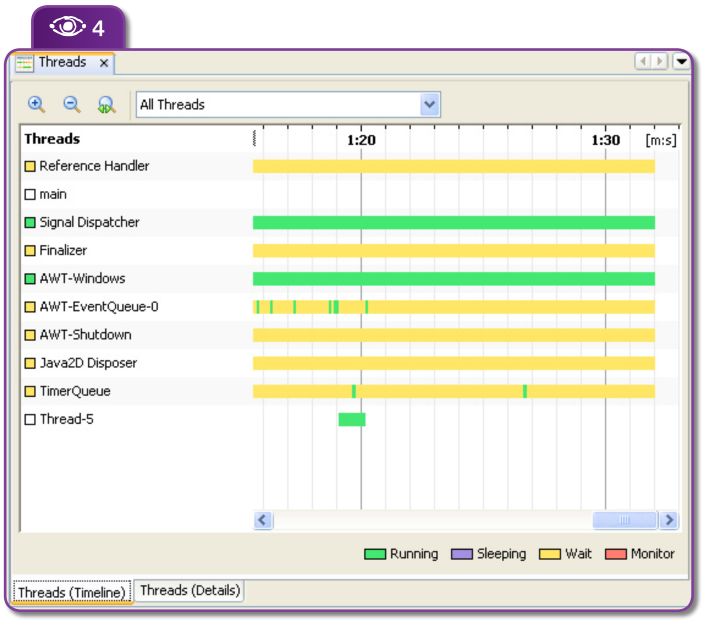


Figure 4 Timeline graph example

CPU Snapshot

The CPU Snapshot captures data on method call chains, times, and invocations when profiling CPU performance, either from the Analyze Performance predefined task or when running a custom CPU profile. The CPU Snapshot is displayed when you click the *Take Snapshot* button ( ) in the control panel. See an example in **Figure 5**. The CPU Snapshot contains the following tabs:

- **Call Tree –** The *Call Tree* tab displays the Calling Context Tree (CCT) showing the method call chain and the time/number of invocations for executing threads and methods in each context. (A context is a unique chain of method calls leading to the method's invocation.)
- **Hot Spots –** The *Hot Spots* tab shows the total execution time and number of invocations for each method, irrespective of the context.
- **Combined –** The *Combined* tab displays the CCT information in the upper half of the window and the Hot Spot data in the lower half.
- **Info –** The *Info* tab displays data on when the snapshot was taken, where it is saved, and the profile settings used.

Memory Snapshot

The Memory Snapshot captures data on object allocation and liveness when profiling memory usage with the Analyze Memory Usage command or when doing memory profiling in a custom profile. Like the CPU Snapshot, the Memory Snapshot is displayed when you click the *Take Snapshot* button ( ) in the control panel.

The Memory Snapshot contains the following tabs:

Memory Results

This tab displays a list of classes (including array classes), with statistics on the total size and number of instances allocated as of the time you took the snapshot (see an example in **Figure 6**). The specific information provided includes:

- *Allocated Objects* is the number of objects that the NetBeans Profiler is actually monitoring. By default this number will be approximately ten percent of the value of total allocated objects. By monitoring only a subset of the created objects the NetBeans Profiler is able to dramatically reduce the overhead it places on the JVM, which then allows your application to run at close to full speed.
- *Live Objects* is the number of the Allocated Objects that are still on the JVM's heap and are therefore taking up memory.
- The two *Live Bytes* columns show the amount of heap memory being used by the Live Objects. One column displays a graph; the other displays text.
- *The Avg. Age value* is calculated using the Live Objects. The age of each object is the number of garbage collections that it has survived. The sum of the ages divided by the number of Live Objects is the Avg. Age.
- The *Generations* value is calculated using the Live Objects. As with Avg. Age, the age of an object is the number of garbage collections it has survived. The Generations value is the number of different ages for the Live Objects. An increasing value for Generations indicates a possible memory leak.

In the *Memory Results* tab, you can right-click any class and select *Show Allocation Stack Traces*. The stack traces for the selected class are displayed in the *Allocation Stack Traces* tab.

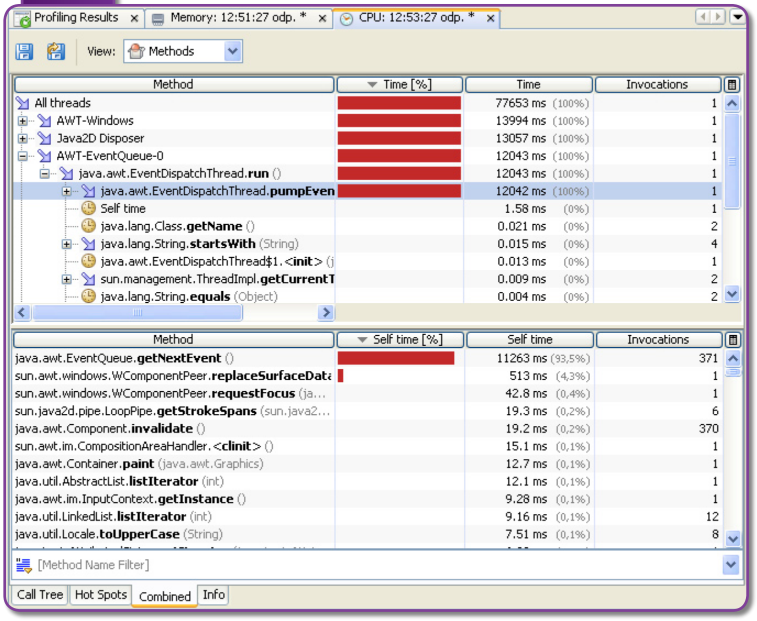
- **Allocation Stack Traces**

This tab displays a reverse call graph with all call paths leading to object allocations for the given class.

- **Info**

The *Info* tab displays data on when the snapshot was taken, where it is saved, and the profile settings used.

5


Profiling Results | Memory: 12:51:27 odp. * | CPU: 12:53:27 odp. *

View: Methods

Method	Time [%]	Time	Invocations
All threads	🟥	77653 ms (100%)	1
AWT-Windows	🟥	13994 ms (100%)	1
Java2D Disposer	🟥	13057 ms (100%)	1
AWT-EventQueue-0	🟥	12043 ms (100%)	1
java.awt.EventDispatchThread.**run** ()	🟥	12043 ms (100%)	1
java.awt.EventDispatchThread.**pumpEven**	🟥	12042 ms (100%)	1
Self time		1.58 ms (0%)	1
java.lang.Class.**getName** ()		0.021 ms (0%)	2
java.lang.String.**startsWith** (String)		0.015 ms (0%)	4
java.awt.EventDispatchThread$1.**<init>** (j		0.013 ms (0%)	1
sun.management.ThreadImpl.**getCurrentT**		0.009 ms (0%)	2
java.lang.String.**equals** (Object)		0.004 ms (0%)	2

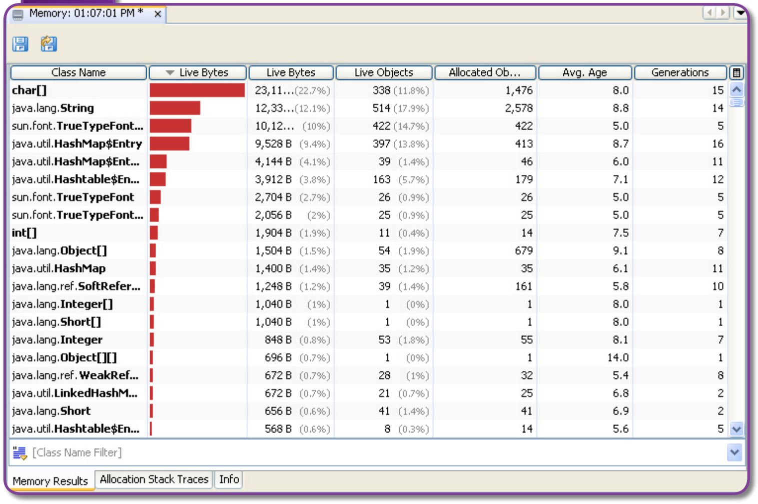
Method	Self time [%]	Self time	Invocations
java.awt.EventQueue.**getNextEvent** ()	🟥	11263 ms (93,5%)	371
sun.awt.windows.WComponentPeer.**replaceSurfaceDat**ɛ	🟥	513 ms (4,3%)	1
sun.awt.windows.WComponentPeer.**requestFocus** (ja...		42.8 ms (0,4%)	1
sun.java2d.pipe.LoopPipe.**getStrokeSpans** (sun.java2...		19.3 ms (0,2%)	6
java.awt.Component.**invalidate** ()		19.2 ms (0,2%)	370
sun.awt.im.CompositionAreaHandler.**<clinit>** ()		15.1 ms (0,1%)	1
java.awt.Container.**paint** (java.awt.Graphics)		12.7 ms (0,1%)	1
java.util.AbstractList.**listIterator** (int)		12.1 ms (0,1%)	1
java.awt.im.InputContext.**getInstance** ()		9.28 ms (0,1%)	1
java.util.LinkedList.**listIterator** (int)		9.16 ms (0,1%)	12
java.util.Locale.**toUpperCase** (String)		7.51 ms (0,1%)	8

[Method Name Filter]

Call Tree | Hot Spots | Combined | Info

Figure 5
CPU Snapshot showing Calling Context Tree information and Hot Spot data

6


Memory: 01:07:01 PM *

Class Name	Live Bytes	Live Bytes	Live Objects	Allocated Ob...	Avg. Age	Generations
char[]	🟥	23,11...(22.7%)	338 (11.8%)	1,476	8.0	15
java.lang.**String**	🟥	12,33...(12.1%)	514 (17.9%)	2,578	8.8	14
sun.font.**TrueTypeFont...**	🟥	10,12... (10%)	422 (14.7%)	422	5.0	5
java.util.**HashMap$Entry**	🟥	9,528 B (9.4%)	397 (13.8%)	413	8.7	16
java.util.**HashMap$Ent...**	🟥	4,144 B (4.1%)	39 (1.4%)	46	6.0	11
java.util.**Hashtable$En...**	🟥	3,912 B (3.8%)	163 (5.7%)	179	7.1	12
sun.font.**TrueTypeFont**	🟥	2,704 B (2.7%)	26 (0.9%)	26	5.0	5
sun.font.**TrueTypeFont...**	🟥	2,056 B (2%)	25 (0.9%)	25	5.0	5
int[]	🟥	1,904 B (1.9%)	11 (0.4%)	14	7.5	7
java.lang.**Object[]**	🟥	1,504 B (1.5%)	54 (1.9%)	679	9.1	8
java.util.**HashMap**	🟥	1,400 B (1.4%)	35 (1.2%)	35	6.1	11
java.lang.ref.**SoftRefer...**	🟥	1,248 B (1.2%)	39 (1.4%)	161	5.8	10
java.lang.**Integer[]**	🟥	1,040 B (1%)	1 (0%)	1	8.0	1
java.lang.**Short[]**	🟥	1,040 B (1%)	1 (0%)	1	8.0	1
java.lang.**Integer**	🟥	848 B (0.8%)	53 (1.8%)	55	8.1	7
java.lang.**Object[][]**	🟥	696 B (0.7%)	1 (0%)	1	14.0	1
java.lang.ref.**WeakRef...**	🟥	672 B (0.7%)	28 (1%)	32	5.4	8
java.util.**LinkedHashM...**	🟥	672 B (0.7%)	21 (0.7%)	25	6.8	2
java.lang.**Short**	🟥	656 B (0.6%)	41 (1.4%)	41	6.9	2
java.util.**Hashtable$En...**	🟥	568 B (0.6%)	8 (0.3%)	14	5.6	5

[Class Name Filter]

Memory Results | Allocation Stack Traces | Info

Figure 6
The Memory Results tab shows statistics on the total size and number of instances allocated

mustang.dev.java.net

Project Mustang (Java SE 6.0)

Profiling example

The following example demonstrates one important feature of the NetBeans Profiler: the ability to quickly identify CPU performance problems. The example is a web application that calculates prime numbers. When attempting to find performance bottlenecks, you typically know which features are running slowly. That allows you to narrow down the search for the bottleneck to a top-level method for that feature. The NetBeans Profiler supports this by allowing you to specify a *root method* for profiling.

1. A profiling session begins by selecting a project as the IDE's Main Project. Then *Profile| Profile Main Project* is chosen from the IDE menu.

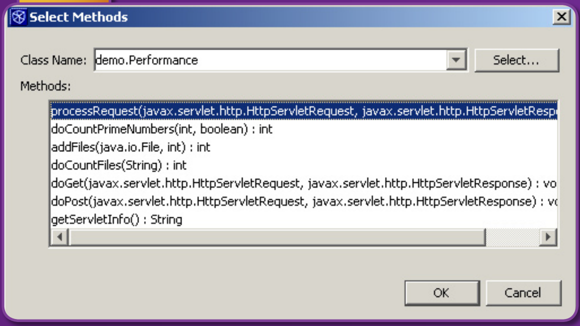
2. The *Select Profiling Task* dialog is displayed.

3. The *Analyze Performance* button is clicked.

4. The *Part of Application* radio button is clicked. Then the *Select* button is used to select the class that contains the *root method*. In this case the class is **demo.Performance** and the method is **processRequest()** – see **Figure 7**. This means that the **demo.Performance.processRequest()** method and all methods that it calls, and all methods that they in turn call (and so on) will be profiled. Starting from **demo.Performance.processRequest()**, the Profiler does analysis of the method call graph to determine which methods need profiling. Only those methods are profiled – the rest of the application will continue to run at full speed with no profiling overhead.

5. Particularly when profiling web or enterprise applications, there are usually large blocks of code that you do not want to profile. In this example, the web server is Tomcat and there is no need to do profiling of Tomcat's code. So in the *Analyze*

7


Select Methods

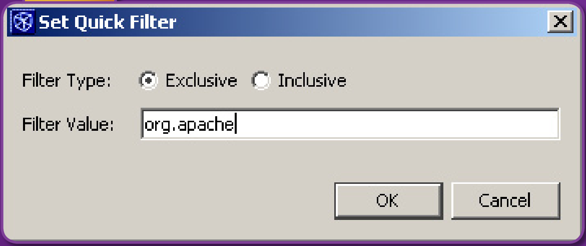
Class Name: demo.Performance

Methods:

```
processRequest(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResp
doCountPrimeNumbers(int, boolean) : int
addFiles(java.io.File, int) : int
doCountFiles(String) : int
doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) : vo
doPost(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse) : vo
getServletInfo() : String
```

Figure 7 Selecting methods to profile

8

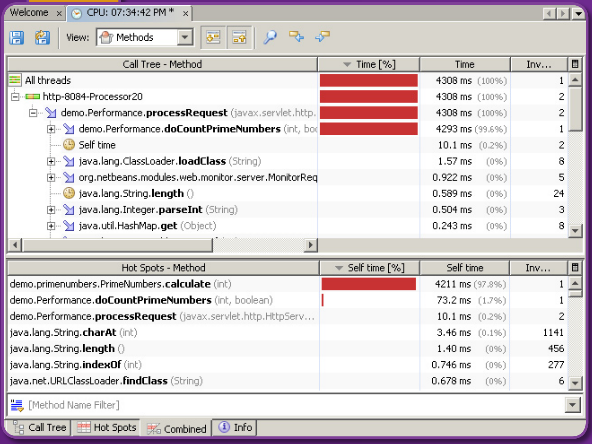

Set Quick Filter

Filter Type: [✓ Exclusive] Inclusive

Filter Value: org.apache

Figure 8 Filtering out methods from apache.org and child packages

9



Call Tree - Method	Time [%]	Time	Inv...
All threads	🟥	4308 ms (100%)	1
http-8084-Processor20	🟥	4308 ms (100%)	2
demo.Performance.**processRequest** (javax.servlet.http.	🟥	4308 ms (100%)	2
demo.Performance.**doCountPrimeNumbers** (int, boo	🟥	4293 ms (99.6%)	1
Self time		10.1 ms (0.2%)	2
java.lang.ClassLoader.**loadClass** (String)		1.57 ms (0%)	8
org.netbeans.modules.web.monitor.server.MonitorReq		0.922 ms (0%)	5
java.lang.String.**length** ()		0.589 ms (0%)	24
java.lang.Integer.**parseInt** (String)		0.504 ms (0%)	3
java.util.HashMap.**get** (Object)		0.243 ms (0%)	8

Hot Spots - Method	Self time [%]	Self time	Inv...
demo.primenumbers.PrimeNumbers.**calculate** (int)	🟥	4211 ms (97.8%)	1
demo.Performance.**doCountPrimeNumbers** (int, boolean)	🟥	73.2 ms (1.7%)	1
demo.Performance.**processRequest** (javax.servlet.http.HttpServ...		10.1 ms (0.2%)	2
java.lang.String.**charAt** (int)		3.46 ms (0.1%)	1141
java.lang.String.**length** ()		1.40 ms (0%)	456
java.lang.String.**indexOf** (int)		0.746 ms (0%)	277
java.net.URLClassLoader.**findClass** (String)		0.678 ms (0%)	6

[Method Name Filter]

Figure 9 An initial snapshot of the performance results

Performance window, the *Quick Filter* is used to specify methods that should *not* be profiled. The string "org.apache" is specified so that all methods in the **org.apache** package (and child packages) will not be profiled – even if they are called from the root method that was selected (see **Figure 8**). This reduces profiling overhead and filters out information that is not relevant.

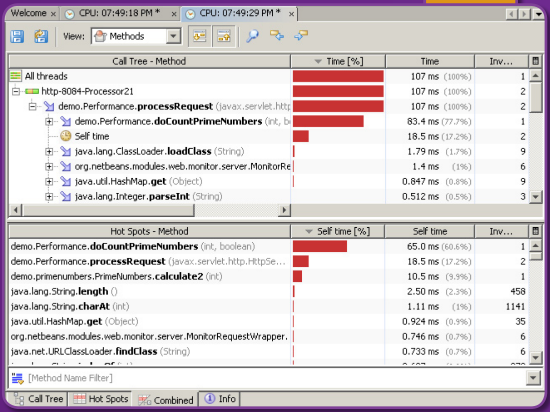
6. Clicking the *Run* button in the *Select Profiling Task* window starts the profiling session. The IDE will start Tomcat and display the web application's *index.jsp* page in a web browser window. At the same time, the Profiler will run in the background.

7. The portion of the web application that causes the *root method* to run is then invoked by interacting with the application's user interface.

8. After the application responds, the *Profile|Take Snapshot of Collected Results* command is selected in the IDE. The Profiler displays the performance results, as illustrated in **Figure 9**.

9. The top window shows the complete method call graph beginning with the *root method*. The bottom window is a flatter depiction; it shows the Hot Spots in the application – those methods that took the most time to execute.

10. To examine and interpret the results, notice that the **processRequest()** method ran for a total of 4308 milliseconds (ms). Note, however, that very little time was spent running the instructions of the



Call Tree - Method	Time [%]	Time	Inv...
All threads		107 ms (100%)	1
http-8084-Processor21		107 ms (100%)	2
demo.Performance.**processRequest** (javax.servlet.http		107 ms (100%)	2
demo.Performance.**doCountPrimeNumbers** (int, b		83.4 ms (77.7%)	1
Self time		18.5 ms (17.2%)	2
java.lang.ClassLoader.**loadClass** (String)		1.79 ms (1.7%)	9
org.netbeans.modules.web.monitor.server.MonitorRe		1.4 ms (1%)	6
java.util.HashMap.**get** (Object)		0.847 ms (0.8%)	9
java.lang.Integer.**parseInt** (String)		0.512 ms (0.5%)	3

Hot Spots - Method	Self time [%]	Self time	Inv...
demo.Performance.**doCountPrimeNumbers** (int, boolean)		65.0 ms (60.6%)	1
demo.Performance.**processRequest** (javax.servlet.http.HttpSe...		18.5 ms (17.2%)	2
demo.primenumbers.PrimeNumbers.**calculate2** (int)		10.5 ms (9.9%)	1
java.lang.String.**length** ()		2.50 ms (2.3%)	458
java.lang.String.**charAt** (int)		1.11 ms (1%)	1141
java.util.HashMap.**get** (Object)		0.924 ms (0.9%)	35
org.netbeans.modules.web.monitor.server.MonitorRequestWrapper.		0.746 ms (0.7%)	6
java.net.URLClassLoader.**findClass** (String)		0.733 ms (0.7%)	6

Figure 10
Profiler results after method optimization

processRequest() method itself – the "self time" for **processRequest()** is only 10.1 ms. The vast majority of the time was spent in methods called by **processRequest()**. The Hot Spots displayed in the bottom window are sorted by "self time." By looking at that list you can see that the **calculate()** method took up 97.8% of the execution time. This is not surprising given the amount of work the **calculate()** method has been given to do *and* the inefficient way it goes about doing that work.

11. To help you decide how your application can be optimized, the NetBeans Profiler helps you identify bottlenecks in your code that were not expected or that will prevent your application from scaling well. From here, it is possible to right-click the **calculate()** entry and choose *Go To Source* in order to examine the source code. As a comparision to **calculate()**'s runtime, the Profiler output of an optimized algorithm in a method called **calculate2()**, is shown in **Figure 10**. Notice that the **processRequest()** method ran for only 107ms and the **calculate2()** method took up less than 10% of the execution time!

Conclusions

The NetBeans Profiler is a powerful tool that provides important information about the runtime behavior of an application. It can be used to identify thread state problems, CPU performance bottlenecks, and memory usage bugs.

Gregg Sporar
(*gregg.sporar@sun.com*) has been a software developer for over twenty years, working on projects ranging from control software for a burglar alarm to 3D graphical user interfaces. He has been using Java since 1998 and his interests include user interfaces, development tools, and performance profiling. He works for Sun Microsystems as a Technical Evangelist on the NetBeans project.

Plug-in Module Quick Start

Extending NetBeans 5

Geertjan Wielenga

This document takes you through the basics of using NetBeans IDE 5.0 to develop NetBeans plug-in modules. You develop NetBeans plug-in modules for one of two reasons:

- **To extend the NetBeans IDE.** You can very easily extend the IDE's functionality with new features. For example, you can write plug-in modules that make your favorite technologies available to the NetBeans IDE.
- **To build an application on top of the NetBeans Platform.** You can use the core of NetBeans as a platform on top of which you develop rich client applications. You can save a lot of development time by reusing features readily available in he platform.

Mainly the first scenario above is covered in this tutorial, although the principles addressed here also apply to the second. Here you will create and install a simple NetBeans plug-in module which will add a new menu

item and a toolbar button to the IDE. When you select the menu item or toolbar button, a **DialogDisplayer**, provided by the NetBeans APIs, with the text "I'm plugged in!" will be shown.

Setting Up a Plug-in Module Project

NetBeans provides a wizard that sets up all the basic files needed for a plug-in module.

Creating a NetBeans Plug-in Module Project

In NetBeans, choose *File|New Project*, and under *Categories* select *NetBeans Plug-in Modules*. NetBeans plug-in module support provides three project types:

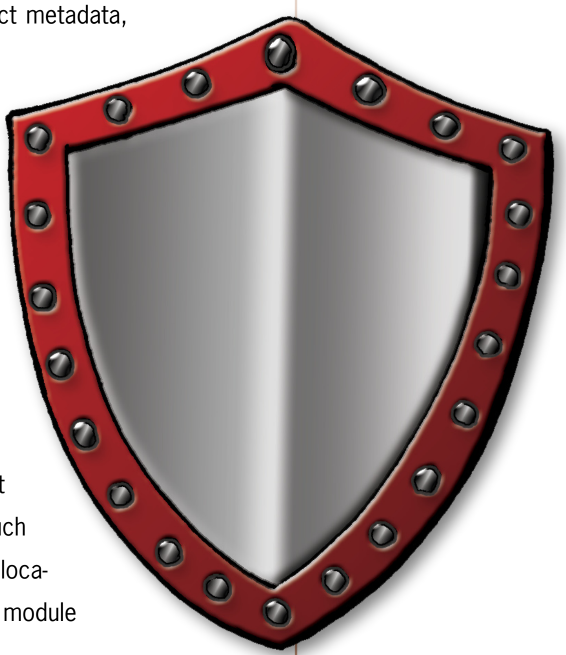
- **Module Project**. Creates a template for a standalone plug-in module.
- **Library Wrapper Module Project**. Creates a plug-in module for an external JAR file required by one or more plug-in modules.
- **Module Suite Project**. Creates a template for a set of interdependent plug-in modules and library wrapper modules, which you want to deploy together.

Select *Module Project* and click *Next*. In the *Name and Location* panel, type "MyFirstModule" in *Project Name*. Change the *Project Location* to any directory on your computer, such as *c:\mymodules*. Leave the *Standalone Module* radio button and the *Set as Main Project* checkbox selected. Click *Next*.

In the *Basic Module Configuration* panel, replace "yourorghere" in *Code Name Base* with "myorg", so that the whole code name base is "org.myorg.myfirstmodule". Notice that the localizing bundle and the XML layer will be stored in the package **org.myorg.myfirstmodule**. These files do the following:

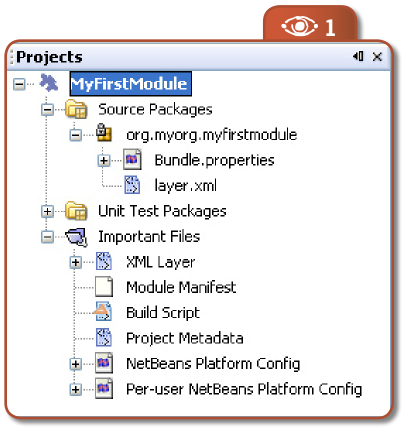
- **Localizing Bundle.** Specifies language-specific strings for internationalization.
- **XML Layer.** Registers items such as menus and toolbar buttons in the NetBeans System Filesystem (see the **sidebar** "Plug-in Module Terms").

Click *Finish*. The IDE creates the MyFirstModule project, containing all of your sources and project metadata, such as the project's Ant build script. The project opens in the IDE. You can view its logical structure in the *Projects* window (Ctrl-1) and its file structure in the *Files* window (Ctrl-2). See **Figure 1**.



In addition to the localizing bundle and the XML layer, the project also includes the following important files:

- **Module Manifest.** Declares that the project is a plug-in module. In addition, it sets some module-specific settings, such as the location of the XML layer, the location of the localizing bundle, and the module version.
- **Build Script.** Provides a place where you can create your own Ant targets and override those that are specified in *nbproject/build-impl.xml*.
- **Project Metadata.** Contains information such as the project's type, contents, platform, classpath, dependencies, and mappings between project commands and targets in Ant scripts.
- **NetBeans Platform Config.** Contains properties used by the IDE or Platform.
- **Per-user NetBeans PlatformConfig** Contains properties specific to your installation of the IDE.



```
Projects
MyFirstModule
  Source Packages
    org.myorg.myfirstmodule
      Bundle.properties
      layer.xml
  Unit Test Packages
  Important Files
    XML Layer
    Module Manifest
    Build Script
    Project Metadata
    NetBeans Platform Config
    Per-user NetBeans Platform Config
```

Figure 1 Logical structure of the new plug-in module

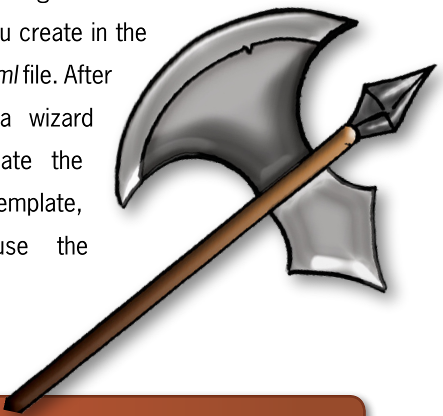
NetBeans Platform netbeans.org/products/platform

For example, if you are sharing the project over VCS, any properties you set in this file are not checked into the repository. You can copy a property from NetBeans Platform Config into this file and give the property different definitions in each file. The definitions in this file take precedence over those in NetBeans Platform Config.

You will not need to modify any of these files during this tutorial. Note that the important files shown before are the logical views of the following files in the Files window: *manifest.mf*, *build.xml*, *nbproject/project.xml*, *nbproject/platform.properties*, and *nbproject/private/platform-private.properties*, respectively.

Creating a Menu Item and Toolbar Button

You use the NetBeans plug-in module file templates to create the basis of the module's functionality. When you use a file template, the IDE registers the item that you create in the *layer.xml* file. After using a wizard to create the file template, you use the



Plug-in module terms

The basic terms used in plug-in module development are as follows:

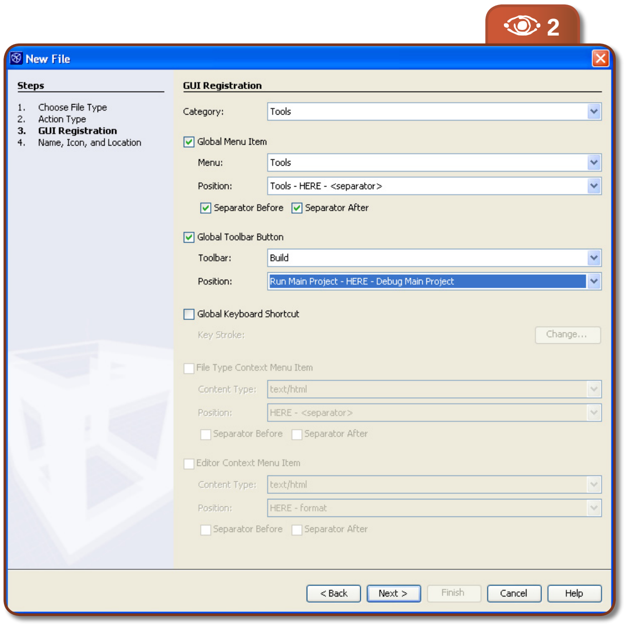
NetBeans Platform. The skeleton application that provides everything most applications need and little of what they don't. The NetBeans Platform provides an application's common requirements – such as menus, document management, and settings – right out of-the-box. Building an application "on top of NetBeans" means that, instead of writing applications from scratch, you only provide the parts of your application that the NetBeans Platform doesn't already have. At the end of the development cycle, you bundle your application with the NetBeans Platform, saving you time and energy and resulting in a solid, reliable application.

System Filesystem. The general registry that contains NetBeans configuration information, built from the *layer.xml* configuration files of the registered modules. NetBeans stores a wide variety of configuration information in the System Filesystem. For example, the System Filesystem contains a folder called *Menu*, which contains subfolders with names such as *File* and *Edit*. These subfolders contain files that represent Java classes which implement the actions that appear in the *File* and *Edit* menus in the IDE.

Plug-in Module. A group of Java classes that provides an application with a specific feature. The Java classes use the *manifest.mf* file to declare the module and the *layer.xml* configuration file to register their functionality in the System Filesystem. In NetBeans terminology, "plug-in" is an adjective while "module" is a noun. There is no discernible difference in meaning between them.

NetBeans APIs. The public interfaces and classes which are available to module writers. They are divided into specific APIs for dealing with different types of functionality. The contents and behavior of the Java source packages and its subpackages, as specified in the API reference documentation, are the APIs.

Module Suite. A group of interdependent modules that are deployed together. The IDE helps you to brand the suite – for example, you can add a splash screen, and also specify the parts of the NetBeans Platform that you don't want your application to provide.


New File

Steps

1. Choose File Type
2. Action Type
3. **GUI Registration**
4. Name, Icon, and Location

GUI Registration

Category: Tools

[✓] Global Menu Item

Menu: Tools

Position: Tools - HERE - <separator>

[✓] Separator Before [✓] Separator After

[✓] Global Toolbar Button

Toolbar: Build

Position: Run Main Project - HERE - Debug Main Project

[ ] Global Keyboard Shortcut

Key Stroke:

[ ] File Type Context Menu Item

Content Type: text/html

Position: HERE - <separator>

[ ] Separator Before [ ] Separator After

[ ] Editor Context Menu Item

Content Type: text/html

Position: HERE - format

[ ] Separator Before [ ] Separator After

Figure 2 Plug-in module GUI Registration

NetBeans API List (see links) to continue developing the module.

Using the Action Wizard

In the *Projects* window, right-click the project node and choose *New>File/Folder*. In the *New File* wizard, choose *NetBeans Module Development* under *Categories*, and *Action* under *File Types*. Click *Next*.

In the *Action Type* panel, accept the defaults and again click *Next*. In the *GUI Registration* panel, select *Global Menu Item*, and then *Global Toolbar Button*. Set the following values:

- **Category**: Tools
- **Menu**: Tools
- **Position**: Tools - HERE - <separator>
- **Toolbar**: Build
- **Position**: Run Main Project - HERE - Debug Main Project

Select *Separator Before and Separator After* in the *Global Menu Item* section. You should now see **Figure 2**.

Note the following about the sections in the *GUI Registration* panel:

- **Category.** Specifies where the action will be located in the *Keymap* section of the *Options* window.
- **Global Menu Item.** Specifies the menu where the action will be registered as a menu item. The position of the menu item within the existing items in the menu can also be set here.
- **Global Toolbar Button.** Specifies the toolbar where the action will be registered as a button. The position of the toolbar button within the existing buttons in the toolbar can also be set in this section.
- **Global Keyboard Shortcut.** Specifies a key stroke that will invoke the action.
- **File Type Context Menu Item.** Specifies the MIME type of the file type where the menu item will appear. The position of the menu item within the existing menu items and its separators can also be set here.
- **Editor Context Menu Item.** Specifies the MIME type for the editor where the menu item will appear. You can also set here the position of the menu item within the existing menu items and its separators.

Click *Next*. In the *Name, Icon, and Location* panel, type "MyFirstAction" in *Class Name* and type "My First Action" in *Display Name*. In *Icon*, browse to a 16x16 pixel icon in your filesystem. For example, you can find some 16x16 pixel icons at the following location within your NetBeans IDE 5.0 installation directory:

```
enterprise2\jakarta-tomcat-5.5.9\
  server\webapps\admin\images
```

Click *Finish*. The IDE creates *MyFirstAction.java* in **org.myorg.myfirstmodule** and opens it in the Source Editor. **Listing 1** shows what you should see.

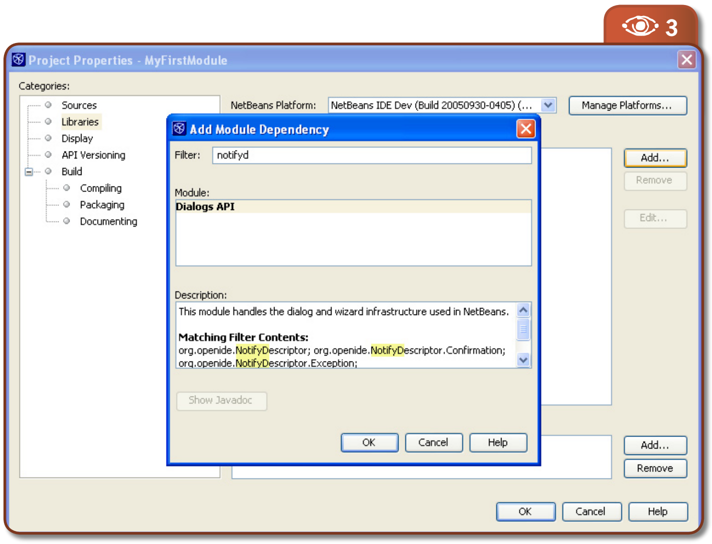
Listing 1. Action class registration in layer.xml

```
<filesystem>
  <folder name="Actions">
    <folder name="Tools">
      <file name="org-myorg-myfirstmodule-MyFirstAction.instance">
         <attr name="instanceClass"
             stringvalue="org.myorg.myfirstmodule.MyFirstAction"/>
      </file>
    </folder>
  </folder>

  <folder name="Menu">
    <folder name="Tools">
      <attr name="org-openide-actions-ToolsAction.instance
          /org-myorg-myfirstmodule-MyFirstAction.shadow"
          boolvalue="true"/>
      <file name="org-myorg-myfirstmodule-MyFirstAction.shadow">
         <attr name="originalFile"
             stringvalue=
               "Actions/Tools/org-myorg-myfirstmodule-MyFirstAction.instance"/>
      </file>
      <attr name="org-myorg-myfirstmodule-MyFirstAction.shadow/Separator1.instance"
          boolvalue="true"/>
    </folder>
  </folder>

  <folder name="Toolbars">
    <folder name="Build">
      <attr name="org-netbeans-modules-project-ui-RunMainProject.shadow
        /org-myorg-myfirstmodule-MyFirstAction.shadow"
        boolvalue="true"/>
      <file name="org-myorg-myfirstmodule-MyFirstAction.shadow">
         <attr name="originalFile"
            stringvalue=
              "Actions/Tools/org-myorg-myfirstmodule-MyFirstAction.instance"/>
      </file>
    <attr name="org-myorg-myfirstmodule-MyFirstAction.shadow
      /org-netbeans-modules-project-ui-DebugMainProject.shadow"
      boolvalue="true"/>
    </folder>
  </folder>
</filesystem>
```

NetBeans API List and documentation

netbeans.org/download/dev/javadoc

As specified in the *GUI Registration* panel, the IDE registers the action class as a menu item and as a toolbar button in the *layer.xml* file. See **Listing 2**.



NetBeans Platform: NetBeans IDE Dev (Build 20050930-0405) (...

Filter: notifyd

Module:
Dialogs API

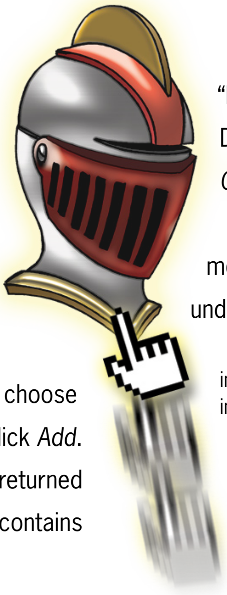
Description:
This module handles the dialog and wizard infrastructure used in NetBeans.

Matching Filter Contents:
org.openide.NotifyDescriptor; org.openide.NotifyDescriptor.Confirmation;
org.openide.NotifyDescriptor.Exception;

Figure 3 Adding a module dependency

In the Source Editor, fill out the **performAction()** method as follows:

```
public void performAction() {
  String msg = "I'm plugged in!";
    NotifyDescriptor d = new NotifyDescriptor.Message(
      msg, NotifyDescriptor.INFORMATION_MESSAGE);
    DialogDisplayer.getDefault().notify(d);
}
```



The line with the calls to **NotifyDescriptor** and **Dialog Displayer** are underlined and marked as an error. This is because the related packages have not been declared yet. In the *Projects* window, right-click the MyFirstModule project node and choose *Properties*. In the *Libraries* pane, click *Add*. Type “notifyd” and notice that the returned list narrows until the package that contains “NotifyDescriptor” is displayed (see **Figure 3**). Click *OK*. The Dialogs API is added to the *Module Dependencies* list. Click *OK* to confirm and exit the *Project Properties* dialog box.

In the Source Editor, click Alt-Shift-F. Two new import statements are added to the top of the source file and the red underlining disappears:

```
import org.openide.DialogDisplayer;
import org.openide.NotifyDescriptor;
```

The plug-in module is now complete. Next, you need to install and use it.

Installing and Using the Plug-in Module

The IDE uses an Ant build script to build and install your plug-in module. The build script is created for you when you create the plug-in module project.

Installing the Plug-in Module

In the *Projects* window, right-click the MyFirstModule project and choose *Install/Reload* in *Target Platform*.

The plug-in module is built and installed in the target platform. The target platform is set in *Tools>NetBeans Platform Manager*. The target platform opens so that you can try out your new plug-in module. The default target IDE or Platform is the installation used by the current instance of the development IDE.

Using the Plug-in Module

In the newly opened IDE’s menu bar, you should see the new menu and menu item, together with the icon you specified in the Action wizard; see **Figure 4**.

Choose the menu item to invoke the **performAction()** method in *MyFirstAction.java*. You should see the **JOptionPane** with its message, as shown in **Figure 5**. Click the toolbar button. It calls the same action and so has the same result. It should look something like **Figure 6**.

Summary

This tutorial showed how to create a simple plug-in module project, including a new menu item and a toolbar button. You also saw how to install and use the new module.

4

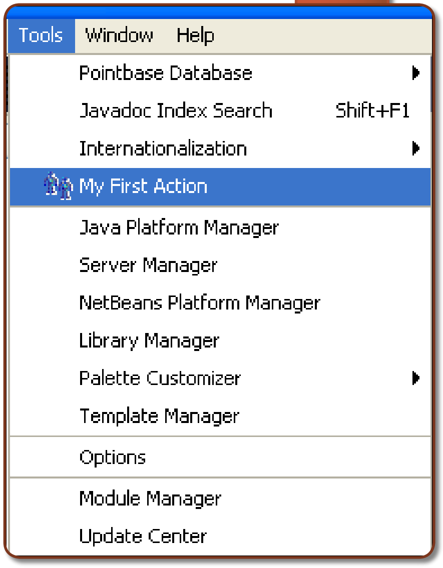
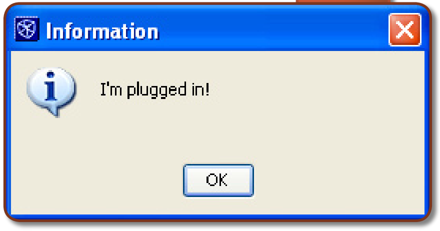


Figure 4
The new menu item for the plug-in module

5



I'm plugged in!

Figure 5
The plug-in module in action

6

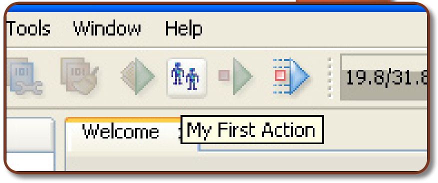
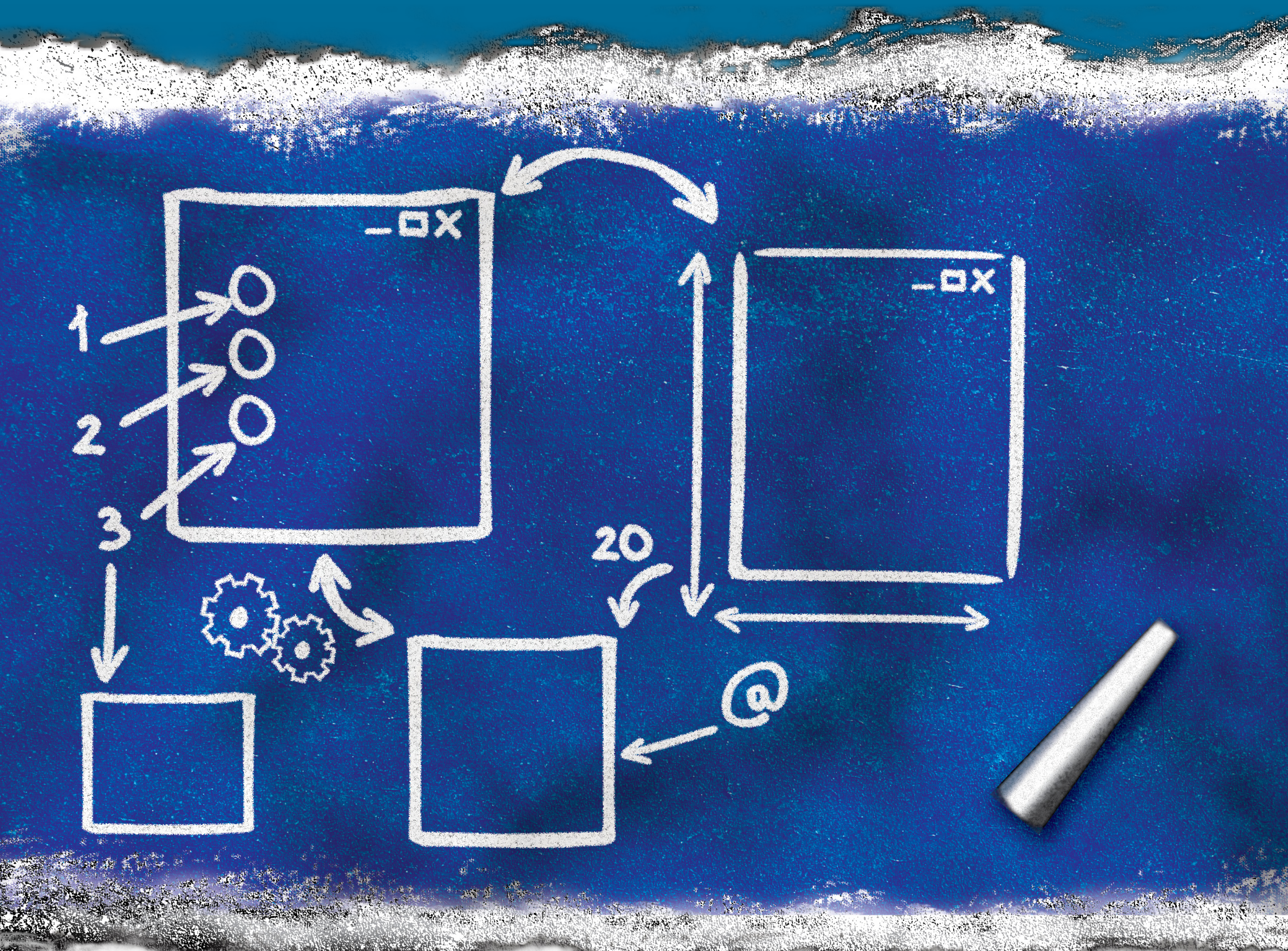


Figure 6
New toolbar button for t

Get to Know GroupLayout

Dissecting the Matisse Core

Tomas Pavek



G**roupLayout** is a new layout manager that was developed as a Swing Labs project in conjunction with Matisse, the new GUI builder in NetBeans 5.0, and is now part of the Mustang project. Though the layout manager was originally designed to suit the GUI builder needs, it is also quite handy for manual coding. This article will help you get up to speed with how **GroupLayout** works and shows you how you can start building GUIs using it, whether you choose to use NetBeans GUI Builder or write your own code.

In this article, I'm going to cover some of the theory behind **GroupLayout** and create a complete example with detailed explanation.

Design principle: independent dimensions

The first thing you need to know about **GroupLayout** is that it works with horizontal and vertical layout separately. This is not that uncommon, but unlike other layout managers **GroupLayout** does not use a single *constraints* object or method to completely specify a component's layout; the layout is defined for each dimension independently.

This might seem a bit unusual at first, but it actually makes things easier because the definition is simpler. When defining the *horizontal* layout, you don't need to worry about the *vertical* dimension, and vice versa. The layout along the horizontal axis is quite independent of the layout along the vertical axis. By focusing just on one dimension at a time you only have to solve half the problem; the other dimension can be solved later. The downside of this approach is that each component needs to be defined twice in the layout. (You'll find out if you forgot to do this, because **GroupLayout** will generate an exception.)

This dimension independence is quite a powerful concept. It is similar to **SpringLayout**, because it provides flexibility other layouts can't offer. We'll get back to this topic later; but first let's see what makes **GroupLayout** *different* from **SpringLayout** and other layout managers.

Layout organization: hierarchical groups

GroupLayout uses two types of arrangements – sequential and parallel, combined with hierarchical composition. These principles are quite basic and well known from Swing.

1. With **sequential** arrangement, the components are simply placed one after another. Just like **BoxLayout** or **FlowLayout** would do along one axis. The position of each component is defined as being relative to the preceding component. This is important for platform-independent layout.

2. The second way places the components in **parallel**, on top of each other in the same space, and aligned along a common reference point. For example, the components can be right-aligned along the horizontal axis, or baseline-aligned along the vertical axis, etc.

Usually, components placed in parallel in one dimension are in a sequence in the other, so they don't overlap. See the examples below.

What makes these two principles powerful is that they can be combined (nested) hierarchically. For this purpose **GroupLayout** defines **layout groups**. A group is either sequential or parallel and may contain components, gaps and other groups. The size of a sequential group is the sum of the sizes of the contained elements, and the size of a parallel group corresponds to the size of the largest element.

Defining a layout means defining how the components should be grouped by combining the sequential and parallel arrangements. This resembles nested panels with **BoxLayout**, but the groups are quite lightweight compared to panels. There is also a difference in the independent dimensions as described above. Panels do nesting in both dimensions at once, while groups can be nested as needed, for each dimension separately.

That's enough theory for now. Let's take a look at how it works in practice with a simple example.

Simple example

Let's start with something really simple, just three components in a row, as shown in **Figure 1**.

We would like to express this layout using groups. Starting with the horizontal axis it's easy to see there is a *sequential group* of three components arranged from left to right. Along the vertical axis there is a *parallel group* of the same three components (at the same coordinate); let's say they are aligned along a baseline. See **Figure 2**.

In pseudo code, the layout specification might look like this (the real code is shown later):

```
horizontal layout = sequential group { c1, c2, c3 }
vertical layout = parallel group (BASELINE) { c1, c2, c3 }
```

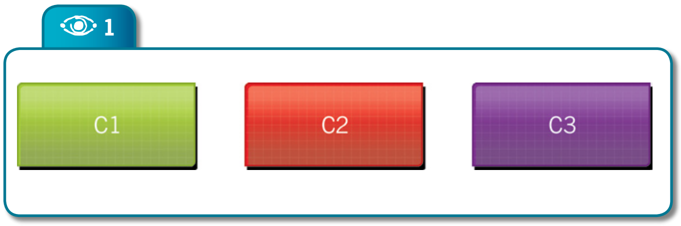


Figure 1
A simple component layout

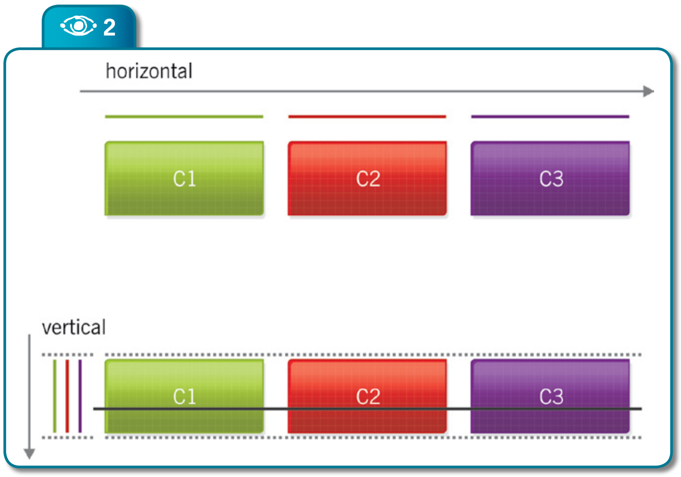


Figure 2
Horizontal and vertical groups

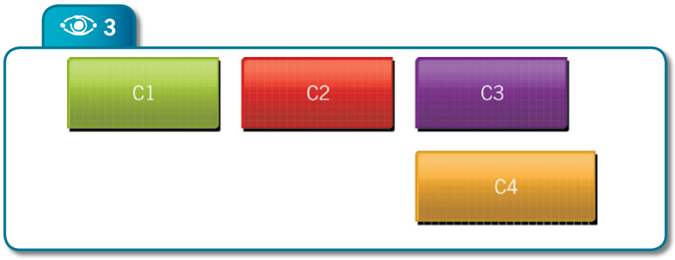


Figure 3
Adding a component to the layout

Note this illustrates a principle I mentioned earlier: components grouped sequentially in one dimension usually form a parallel group in the orthogonal dimension.

Now let's add one more component, C4 (**Figure 3**). Along the horizontal axis the new component forms a parallel group with C3 (because it occupies the same horizontal space as C3), let's say we want the components left aligned. Along the vertical axis C4 forms a sequential group with the original parallel group of the three components (see **Figure 4**).

In pseudo code, the layout specification now looks like this:

```
horizontal layout = sequential group {
   c1, c2, parallel group (LEFT) { c3, c4 } }
vertical layout = sequential group {
     parallel group (BASELINE) { c1, c2, c3 }, c4 }
```

Now that you understand the principle of groups, you know the most important thing about designing layouts with **GroupLayout**. There are some more details to explain, though: how to add gaps, how to define resize behavior, how to write real code, etc.

Gaps

A gap can be thought of as an invisible component of a certain size. Gaps of arbitrary size can be added to groups just like components or other groups. Using gaps you can precisely control the distance between components or from the container border.

GroupLayout also defines a symbolic *default gap* that corresponds to a *preferred distance* between neighboring components (or between a component and container border). The size of such a gap is not defined by an explicit number, but computed dynamically based on the look-and-feel the

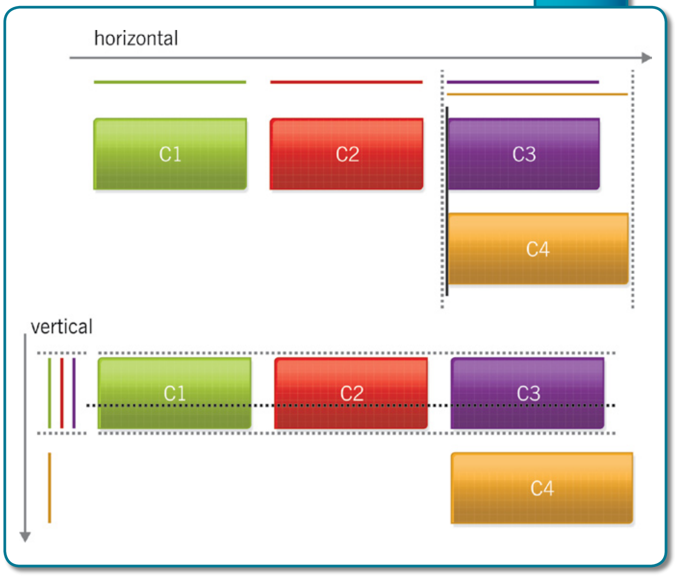


Figure 4
Left aligning components

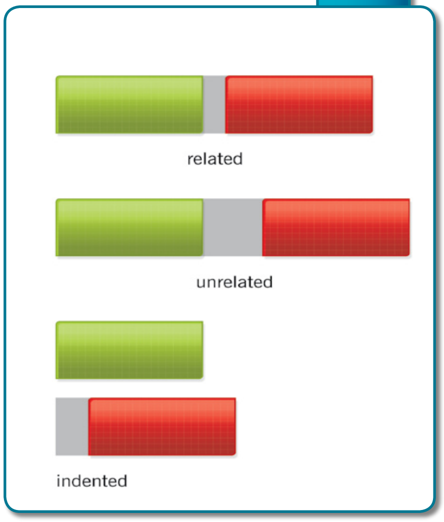


Figure 5
Types of component gaps

application is using (the **LayoutStyle** class is used for this).

There are two advantages to using **preferred gaps**: you don't have to specify the pixel sizes of the gaps, and they automatically adjust to the environment the UI runs in, reflecting the actual platform guidelines.

GroupLayout distinguishes between (a) the preferred gap between two components and (b) the preferred gap between a component and the container border. There are corresponding methods in the **GroupLayout** API for adding these gaps (**addPreferredGap()** and **addContainerGap()**). There are three types of component gaps: **related**, **unrelated** and **indented** (see **Figure 5**). The **LayoutStyle** class defines corresponding constants (to be used as the first parameter of **addPreferredGap()**): **RELATED**, **UNRELATED** and **INDENT**. The difference between related and unrelated gaps is just in size (the distance between unrelated components is a bit bigger). Indented represents a preferred horizontal distance of two components, where one of them is positioned underneath the second with an indent.

To make things easier, **GroupLayout** can insert gaps automatically. If you don't add your own gaps explicitly, it adds the *related* preferred gaps for you. This is not the default behavior, you have to turn this feature on by invoking **setAutocreateGaps(true)** and **setAutocreateContainerGaps(true)** on the layout. Then you'll get correct spacing almost for free!

How to write code

Now, let's take a look at the actual code to create the layout described above. We'll assume we have a container named **panel** and four components (**c1**, **c2**, **c3**, **c4**) which are already set up. First, we create a new **GroupLayout** object and associate it with the panel:

```
GroupLayout layout = new GroupLayout(panel);
panel.setLayout(layout);
```

We specify automatic gap insertion:

```
layout.setAutocreateGaps(true);
layout.setAutocreateContainerGaps(true);
```

Finally, we define groups and add the components. We establish a root group for each dimension using **setHorizontalGroup()** and **setVerticalGroup()**. Groups are created via **createSequentialGroup()** and **createParallelGroup()**. Components are added to groups by using a variant of the **add()** method.

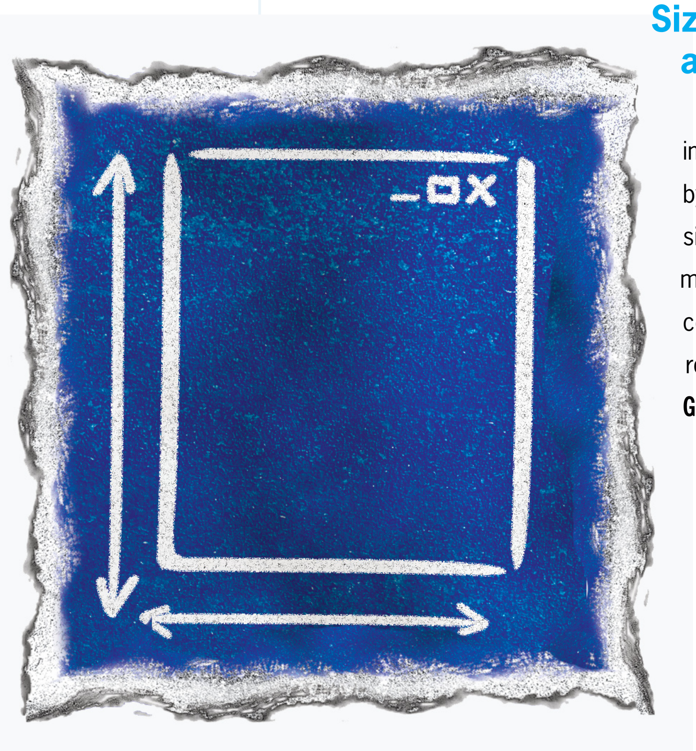
```
layout.setHorizontalGroup(layout.createSequentialGroup()
        .add(c1)
        .add(c2)
        .add(layout.createParallelGroup(GroupLayout.LEADING)
          .add(c3)
          .add(c4))
  );
```

```
layout.setVerticalGroup(layout.createSequentialGroup()
        .add(layout.createParallelGroup(GroupLayout.BASELINE)
            .add(c1)
            .add(c2)
            .add(c3))
        .add(c4)
);
```

Note that default alignment must be specified for parallel groups. It can be one of the following constants defined in the **GroupLayout** class: **LEADING**, **TRAILING** and **CENTER**. These constants are used for both dimensions; in the horizontal dimension **LEADING** means "left", while in the vertical dimension it means "top". Similarly **TRAILING** maps to "right" or "bottom". The **BASELINE** alignment is valid only in the vertical dimension.

Some notes about the code:

- Components are not added to the container directly, they are added to groups. **GroupLayout** adds the components to the container automatically for you.
- Note the chained calls of the **add()** method used to fill the groups. **add()** always returns the group on which it is called. Thanks to this you don't need to use local variables to hold the groups.
- It is a good idea to indent the code so it is easy to see the hierarchical structure of the groups. Give each component a new line, add one level of indent for each new group in the hierarchy. A decent source editor will help you with pairing the parenthesis to close the **createXXXGroup()** methods. By following these simple rules it is easier to add a new component, or remove an existing one.



Size of components and resizability

The size of each component in a **GroupLayout** is constrained by three values; minimum size, preferred size and maximum size. These sizes control how the component resizes within the layout. The **GroupLayout.add(...)** method allows the size constraints to be specified. There is no limit on the number of resizable components in a layout.

If not specified explicitly, the layout asks the component for its default sizes (by using the component's **getMinimumSize()**, **getPreferredSize()** and **getMaximumSize()** methods). Thus you don't need to specify anything for most of the components, e.g. to make **JTextField** resizable or **JButton** fixed, because the components have the desired resizing behavior as default. On the other hand, you can override the default behavior. For example you can make a **JTextField** fixed or **JButton** resizable.

GroupLayout defines constants that provide precise control over resize behavior. They can be used as parameters in the **add(Component comp, int min, int pref, int max)** method. Here are two examples:

1. To force a component to be resizable (allow shrinking and growing):

```
group.add(component, 0, GroupLayout.DEFAULT_SIZE,
  Short.MAX_VALUE) ...
```

This allows the component to resize between zero size (minimum) to any size (**Short.MAX_VALUE** as maximum size means "infinite"). If we wanted the component not to shrink below its default minimum size, we'd use **GroupLayout.DEFAULT_SIZE** instead of **0** in the second parameter.

2. To make a component have a fixed size (suppress resizing):

```
group.add(component, GroupLayout.PREFERRED_SIZE,
  GroupLayout.DEFAULT_SIZE,GroupLayout.PREFERRED_SIZE)
...
```

In these examples the initial size of the component is not altered; its default size is the component's preferred size. If we wanted a specific size for the component, we would specify it in the second parameter instead of using **GroupLayout.DEFAULT_SIZE**.

Resizable gaps

Specifying size and resizability applies to gaps as well, including the preferred ones. For example, you can specify a preferred gap between two components that acts like a *spring* pushing the components away from each other (to the opposite sides of the container). The preferred distance of the two components is only used as the minimum size of the gap. See the following snippet:

```
layout.createSequentialGroup()
  .add(c1)
  .addPreferredGap(LayoutStyle.RELATED,
    GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
  .add(c2);
```

Justified layout

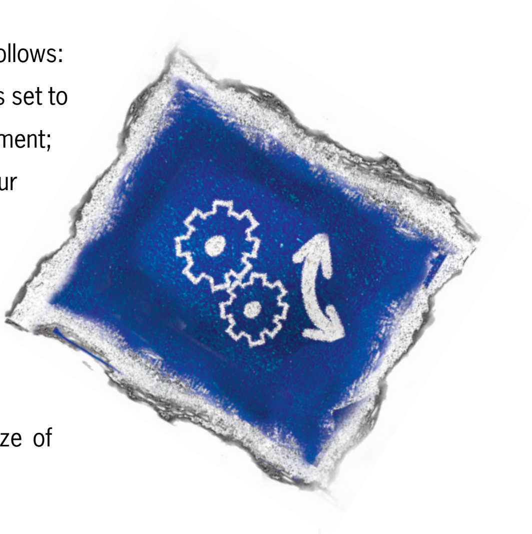
Resizable elements placed in a parallel group are stretched to fill the space of the group determined by the largest element in the group; so they end up aligned with the same size. **GroupLayout** also provides control over whether the enclosing parallel group itself should resize. If group resizing is suppressed, it prevents the contained elements from growing over the preferred size of the group. This way you can make a block of components align on both sides, or constrain individual components to have the same size.

Let's try to achieve the same size for two components from our example (**c3** and **c4** in the horizontal dimension):

```
layout.createParallelGroup(GroupLayout.LEADING, false)
    .add(c3, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
    .add(c4, GroupLayout.DEFAULT_SIZE,
        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE);
```

The underlying mechanism works as follows:

1. The size of the parallel group is set to the preferred size of the largest element; so to the preferred size of **c4** in our example.
2. Resizable elements are stretched to the size of the group. In our example, only **c3** is effectively stretched, the size of **c4** already corresponds to the size of the group.



As a result, **c3** and **c4** would have the same width. The components would not resize further because the parallel group itself is not resizable (see **Figure 6**).

A question for attentive readers: Why do we define both components in the parallel group as resizable in this example? It seems enough to have just **c3** resizable since **c4** is not stretched anyway...

The answer is simple: because of platform and localization independence. Otherwise we would have to rely on the fact that the **c4** component is always bigger than **c3**. But this may change when the application runs on a different platform or is translated to another language. By having both components resizing, they adjust to each other, no matter which one is bigger at the moment.

Same component size

The previous case is special because the components are in the same parallel group. But what if we wanted unrelated components to have the same size? Clearly, the same size can't always be ensured by grouping. The *OK* and *Cancel* buttons in a row at the bottom of a dialog are a good example. For this purpose **GroupLayout** provides a **linkSize** method. This method allows the size of arbitrary components to be linked, regardless of where they are placed. The resulting size of

6

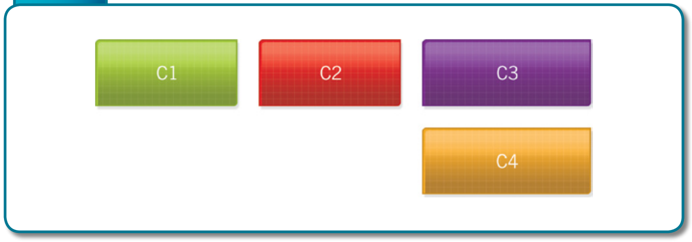


Figure 6
Components C3 and C4 will have the same width

the linked components is set according to the largest component. For example:

```
layout.linkSize(new Component[] {
   c3, c4 }, GroupLayout.HORIZONTAL);
```

Note that in this example the size is linked selectively for the horizontal dimension.

Next I will show how to create a layout for a sample dialog, with detailed explanation of each step, illustrations and code samples.

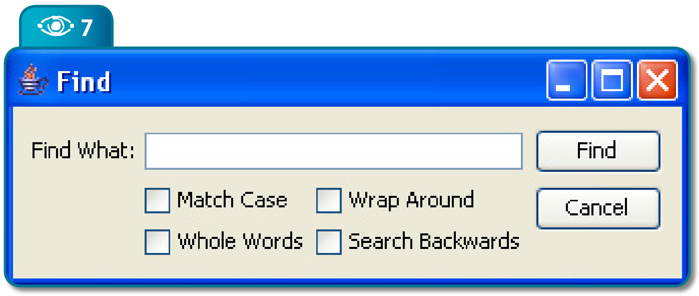


Figure 7 Example dialog to show GroupLayout in practice

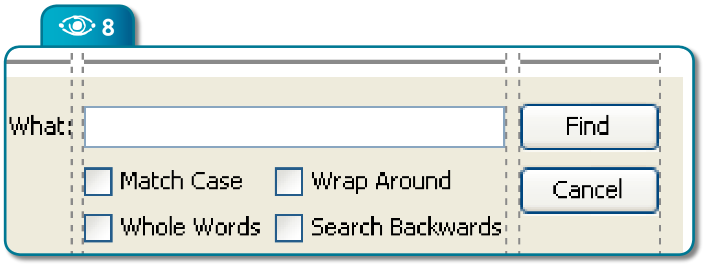


Figure 8 Breaking the layout into groups

A real example

Having provided the theory behind **GroupLayout**, now it's time to try it on a real example. So let's create a layout for a sample dialog, shown in **Figure 7**.

A good way to analyze the layout is to use top-down decomposition. I'll describe the procedure that proved useful to me; step by step to make it clear. Once you get used to "group thinking", you'll be able to proceed much faster.

Horizontal layout

Examining the horizontal dimension *from left to right,* we can see there are three groups in a sequence. The first one is actually not a group, just a component – the label. The second one is a group containing the text field and the checkboxes (we'll decompose it later). And the third is a group of the two buttons. As illustrated in **Figure 8**.

Let's sketch out the sequential group in code. Note that **GroupLayout.LEADING** corresponds to left alignment in the horizontal dimension. Also note we don't specify gaps, assuming the **gap** auto-insertion feature is turned on.

```
layout.setHorizontalGroup(layout.createSequentialGroup()
   .add(label)
   .add(layout.createParallelGroup(GroupLayout.LEADING))
   .add(layout.createParallelGroup(GroupLayout.LEADING))
);
```

Now let's decompose the group in the middle. This is the hardest one. There's a text field in parallel with a sequence of two parallel groups each containing two checkboxes. See **Figure 9**.

Let's add the corresponding code:

```
layout.setHorizontalGroup(layout.createSequentialGroup()
   .add(label)
   .add(layout.createParallelGroup(GroupLayout.LEADING)
      .add(textField)
      .add(layout.createSequentialGroup()
         .add(layout.createParallelGroup(
            GroupLayout.LEADING)
            .add(caseCheckBox)
            .add(wholeCheckBox))
         .add(layout.createParallelGroup(
            GroupLayout.LEADING)
            .add(wrapCheckBox)
            .add(backCheckBox))))
   .add(layout.createParallelGroup(GroupLayout.LEADING))
);
```

Note we want the text field to be resizable, but that happens automatically since **JTextField** returns the right maximum size by default.

The remaining group on the right is trivial: it contains just two buttons. We don't even need the picture. Here's the code:

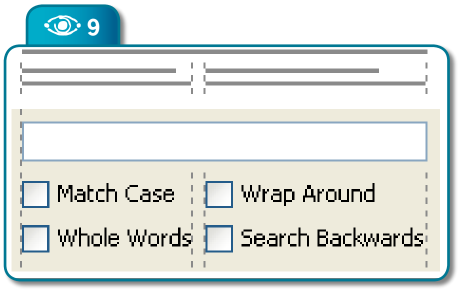


Figure 9 Decomposing the middle group

```
layout.setHorizontalGroup(layout.createSequentialGroup()
    .add(label)
    .add(layout.createParallelGroup(
   GroupLayout.LEADING)
      .add(textField)
      .add(layout.createSequentialGroup()
        .add(layout.createParallelGroup(
          GroupLayout.LEADING)
          .add(caseCheckBox)
          .add(wholeCheckBox))
        .add(layout.createParallelGroup(
          GroupLayout.LEADING)
          .add(wrapCheckBox)
          .add(backCheckBox))))
    .add(layout.createParallelGroup(GroupLayout.LEADING)
      .add(findButton)
      .add(cancelButton))

);
```

And finally we'd like the buttons to have always the same size, so let's link them:

```
layout.linkSize(new Component[] {findButton, cancelButton},
  GroupLayout.HORIZONTAL);
```

Now we are done with the horizontal dimension. Let's switch to the vertical dimension. From now we'll only need to think about the y axis.

Vertical layout

In the vertical dimension, we examine the layout from *top to bottom*. We definitely want all the components on the first line aligned on the baseline. So along the vertical axis there is a sequence of the baseline group, followed by a group of the remaining components. See **Figure 10**.

Let's sketch out the code. First, we need to define two parallel groups. Note that **GroupLayout.LEADING** corresponds to the top alignment in the vertical dimension.

```
layout.setVerticalGroup(layout.createSequentialGroup()
  .add(layout.createParallelGroup(
    GroupLayout.BASELINE))
  .add(layout.createParallelGroup(GroupLayout.LEADING))
);
```

We can fill the baseline group right away:

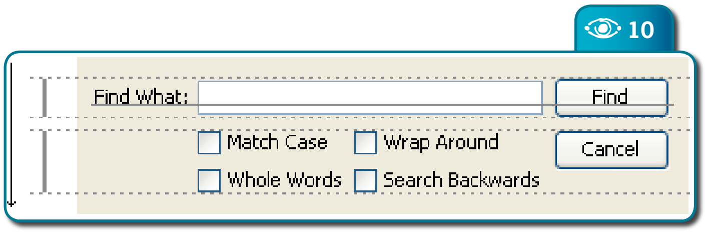


Figure 10
Grouping components top to bottom

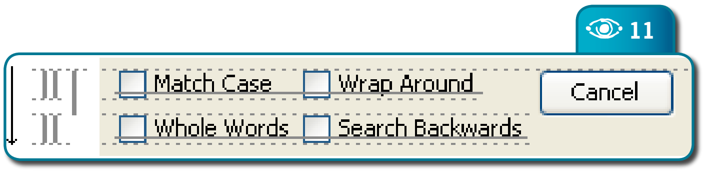


Figure 11
Second parallel group

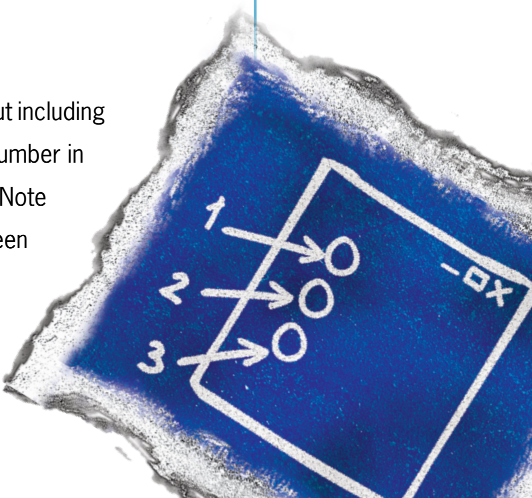
```
layout.setVerticalGroup(layout.createSequentialGroup()
  .add(layout.createParallelGroup(GroupLayout.BASELINE)
    .add(label)
    .add(textField)
    .add(findButton))
  .add(layout.createParallelGroup(GroupLayout.LEADING))
);
```

Now let's look at the bottom group. Note the Cancel button is not on a shared baseline with the checkboxes; it is aligned at the top. So the second parallel group comprises the button and a sequential group of two baseline groups with checkboxes (see **Figure 11**).

The corresponding code looks as follows:

```
layout.etVerticalGroup(layout.createSequentialGroup()
  .add(layout.createParallelGroup(
    GroupLayout.BASELINE)
    .add(label)
    .add(textField)
    .add(findButton))
  .add(layout.createParallelGroup(GroupLayout.LEADING)
    .add(layout.createSequentialGroup()
      .add(layout.createParallelGroup(
        GroupLayout.BASELINE)
.add(caseCheckBox)
.add(wrapCheckBox))
.add(layout.
  createParallelGroup(GroupLayout.BASELINE)
.add(wholeCheckBox)
.add(backCheckBox)))
    .add(cancelButton))
);
```

So, that's it. We've created a complete layout including resize behavior without specifying a single number in pixels. That's a true cross platform layout! Note that we don't need to specify gaps between components; we get correct spacing automatically and according to the look-



Listing 1. *Find.java*: Complete code for the example dialog

```
import java.awt.Component;
import javax.swing.*;
import org.jdesktop.layout.*;

public class Find extends JFrame {
  public Find() {
    JLabel label = new JLabel(“Find What:”);;
    JTextField textField = new JTextField();
    JCheckBox caseCheckBox =
      new JCheckBox(“Match Case”);
    JCheckBox wrapCheckBox =
      new JCheckBox(“Wrap Around”);
    JCheckBox wholeCheckBox =
      new JCheckBox(“Whole Words”);
    JCheckBox backCheckBox =
      new JCheckBox(“Search Backwards”);
    JButton findButton = new JButton(“Find”);
    JButton cancelButton =
      new JButton(“Cancel”);

    // remove redundant default border of
    // check boxes - they would hinder
    // correct spacing and aligning (maybe
    // not needed on some look-and-feels)

    caseCheckBox.setBorder(BorderFactory.
      createEmptyBorder(0, 0, 0, 0));
    wrapCheckBox.setBorder(BorderFactory.
      createEmptyBorder(0, 0, 0, 0));
    wholeCheckBox.setBorder(BorderFactory.
     createEmptyBorder(0, 0, 0, 0));
    backCheckBox.setBorder(BorderFactory.
     createEmptyBorder(0, 0, 0, 0));

    GroupLayout layout = new GroupLayout(
      getContentPane());
    getContentPane().setLayout(layout);
    layout.setAutocreateGaps(true);
    layout.setAutocreateContainerGaps(true);

    layout.setHorizontalGroup(
      layout.createSequentialGroup()
      .add(label)
      .add(layout.createParallelGroup(
      GroupLayout.LEADING)
        .add(textField)
        .add(layout.createSequentialGroup()
          .add(layout.createParallelGroup(
             GroupLayout.LEADING)
            .add(caseCheckBox)
            .add(wholeCheckBox))
          .add(layout.createParallelGroup(
             GroupLayout.LEADING)
            .add(wrapCheckBox)
            .add(backCheckBox))))
      .add(layout.createParallelGroup(
         GroupLayout.LEADING)
        .add(findButton)
        .add(cancelButton))
    );
    layout.linkSize(new Component[] {
      findButton, cancelButton },
         GroupLayout.HORIZONTAL);

    layout.setVerticalGroup(
      layout.createSequentialGroup()
      .add(layout.createParallelGroup(
       GroupLayout.BASELINE)
        .add(label)
        .add(textField)
        .add(findButton))
      .add(layout.createParallelGroup(
        GroupLayout.LEADING)
        .add(layout.createSequentialGroup()
          .add(layout.createParallelGroup(
          GroupLayout.BASELINE)
            .add(caseCheckBox)
            .add(wrapCheckBox))
          .add(layout.createParallelGroup(
          GroupLayout.BASELINE)
            .add(wholeCheckBox)
            .add(backCheckBox)))
        .add(cancelButton))
    );

    setTitle(“Find”);
    pack();
    setDefaultCloseOperation(
      WindowConstants.EXIT_ON_CLOSE);
  }

  public static void main(String args[]) {
    java.awt.EventQueue.invokeLater(
      new Runnable() {
      public void run() {
        try {
          UIManager.setLookAndFeel(UIManager.
           getSystemLookAndFeelClassName());
        } catch (Exception ex) {
          ex.printStackTrace();
        }
        new Find().setVisible(true);
      }
    });
  }
}
```

Tomas Pavek *(tomas.pavek@sun.com)* is a software engineer at Sun Microsystems who has been working on the NetBeans IDE since 2000. He is the lead developer of the GUI builder in NetBeans, now primarily focused on the project Matisse. Tomas lives in Prague, Czech Republic.

and-feel guidelines. **Listing 1** shows the complete code for the Find dialog (it is available for download in the magazine's website).

You can compile this file and run it against the Swing Layout Extensions library. Try resizing the dialog horizontally to see how the layout automatically adjusts to the new size.

Conclusions

Now you should know enough about **GroupLayout** to start using it! The easiest way to test **GroupLayout** is to use it with NetBeans 5.0 where it is bundled (just make sure "Swing Layout Extensions" library is added in the libraries of the working project). The layout manager and related extensions are hosted on *swing-layout.dev.java.net as one of the Swing Labs projects.*

netbeans.org/community/magazine/code/Find.Java

NetBeans is Everywhere

at JavaOne 2006

Don't miss out on Labs, Sessions and BOFs covering IDE features, Matisse, Plug-in Modules, Wireless Development, Enterprise pack, success cases and more.

NetBeans Related Hands-On Labs

May 16		
2:00 PM	LAB-8105	NetBeans Enterprise Pack: Basic UML Modeling
May 17		
2:45 PM	LAB-5250	Building Production Quality GUI Applications using NetBeans 5.0 GUI Builder (Matisse)
May 18		
9:45 AM	LAB-4255	AJAX: Riding the Web Application Horse a Little Further
11:30 AM	LAB-5106	NetBeans 5.0: Plug-in Development
2:45 PM	LAB-8125	NetBeans Enterprise Pack: BPEL
May 19		
10:45 AM	LAB-5120	Application Monitoring and Management with NetBeans 5.0
3:45 PM	LAB-6205	Mobile Applications: Visualize it, Build it, Deploy it to Many Devices

NetBeans-Related Sessions and BOFs

May 16		
5:45 PM	TS-3361	Java EE 5 Platform: Even Easier With Tools
9:30 PM	BOF-2496	Building Development Tools on Top of the NetBeans IDE
10:30 PM	BOF-0678	Meet the "Java Posse"
May 17		
11:00 AM	TS-4255	MHP/OCAP iTV Applications In a Nutshell
12:15 PM	TS-4916	Creating Professional Swing UIs Using Matisse GUI Builder
1:30 PM	TS-1293	Best Practices for Building Optimized Wireless Solutions for Web Services
2:45 PM	TS-4589	Good Morning, Buenos Dias, Dobry Den: Mobile Internationalization in Action
8:30 PM	BOF-2340	Creating NetBeans Plug-ins for Integration With JavaServer Faces, Hibernate, Spring, and EJB™ 3.0 Technology
9:45 PM	TS-1387	Twelve Reasons to Use NetBeans Software: Episode 2
May 18		
9:45 AM	TS-1278	Creating and Deploying Custom Jackpot Queries and Transformers
1:30 PM	TS-1878	Debugging Across Tiers: Advanced Techniques
2:45 PM	TS-1549	Debugging and Profiling J2EE /Java EE 5 Platform-Based Applications
7:30 PM	BOF-2559	Discovery and Dependency Injection Patterns in Modular Architectures
7:30 PM	BOF-2807	Java Persistence API in the NetBeans IDE
8:30 PM	BOF-2417	Memory Leaks in Jav Technology-Based Applications: Different Tools for Different Types of Leaks
9:30 PM	BOF-2835	Creating an IDE for your favorite web framework
9:30 PM	BOF-2461	Use the Tools to Generate Mobile Business!
10:30 PM	BOF-2031	Experience Self-Organized and Ad Hoc Collaborative Development With the NetBeans IDE