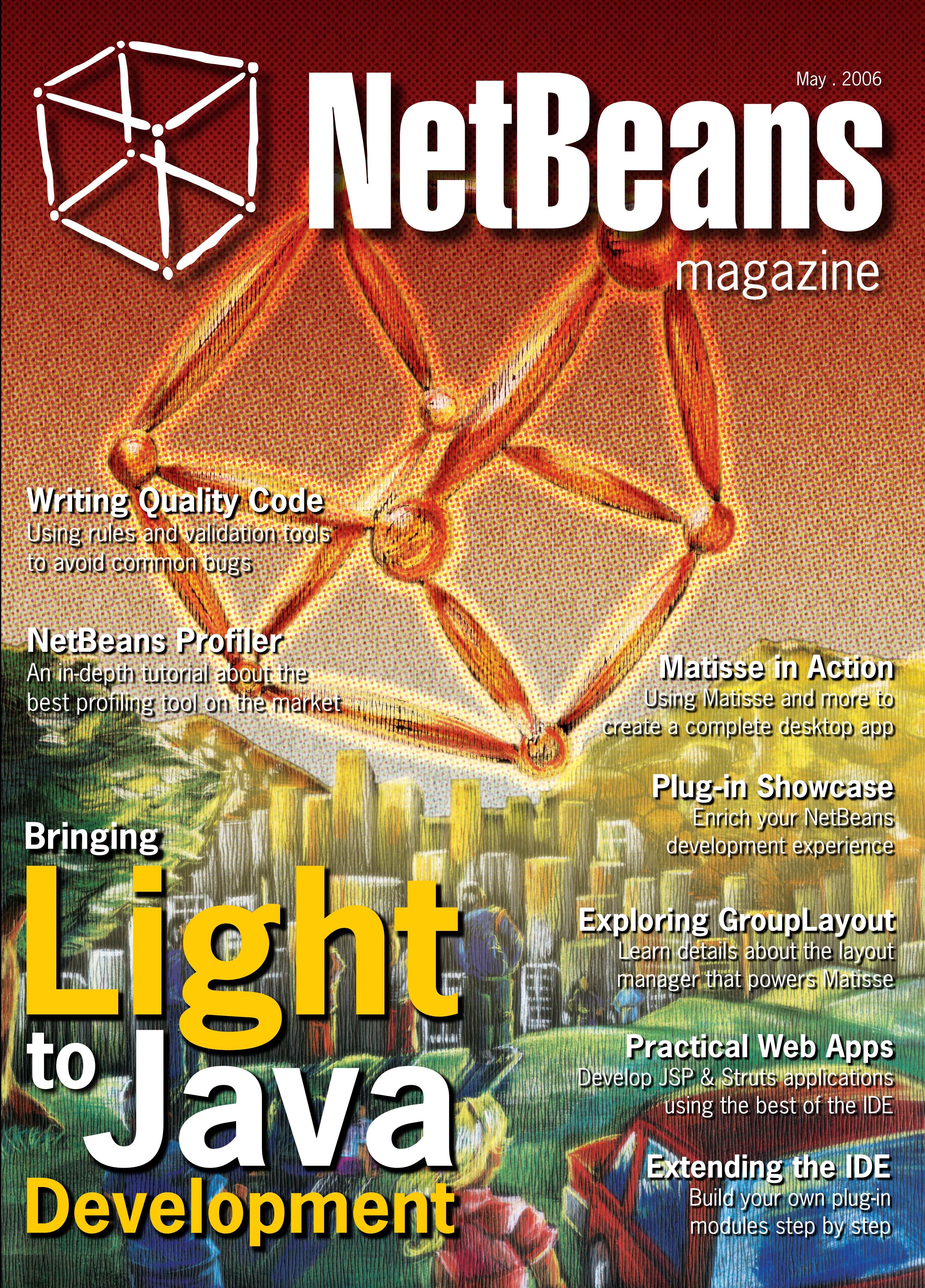
May . 2006

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

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*)
pH DESIGN WWW.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.

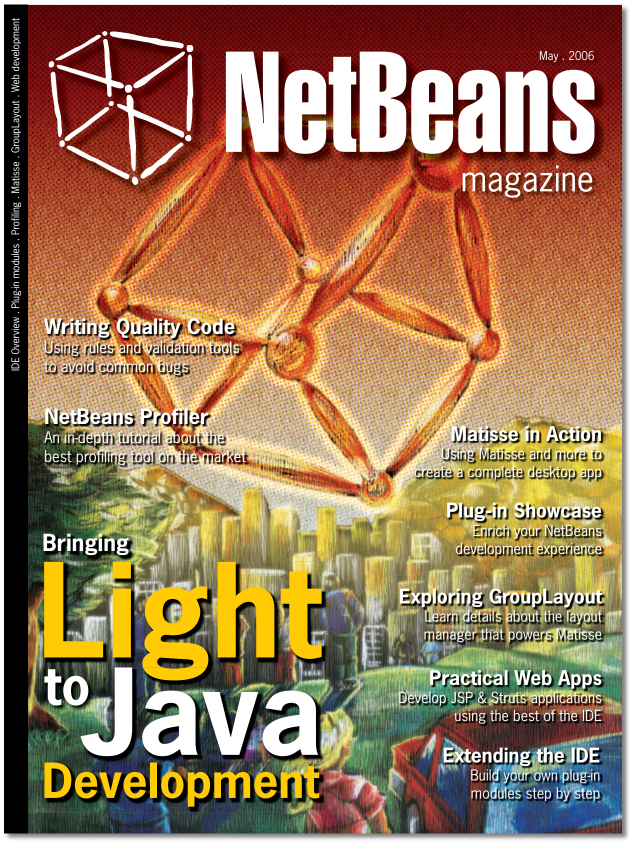
This special first edition of NetBeans Magazine showcases a wide selection of IDE and extension features, from desktop and web development to plug-in module creation. Beginners will learn how to develop a complete desktop application using Matisse and other IDE facilities. Seasoned programmers will also 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

Exploring GroupLayout
Learn details about the layout manager that powers Matisse

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

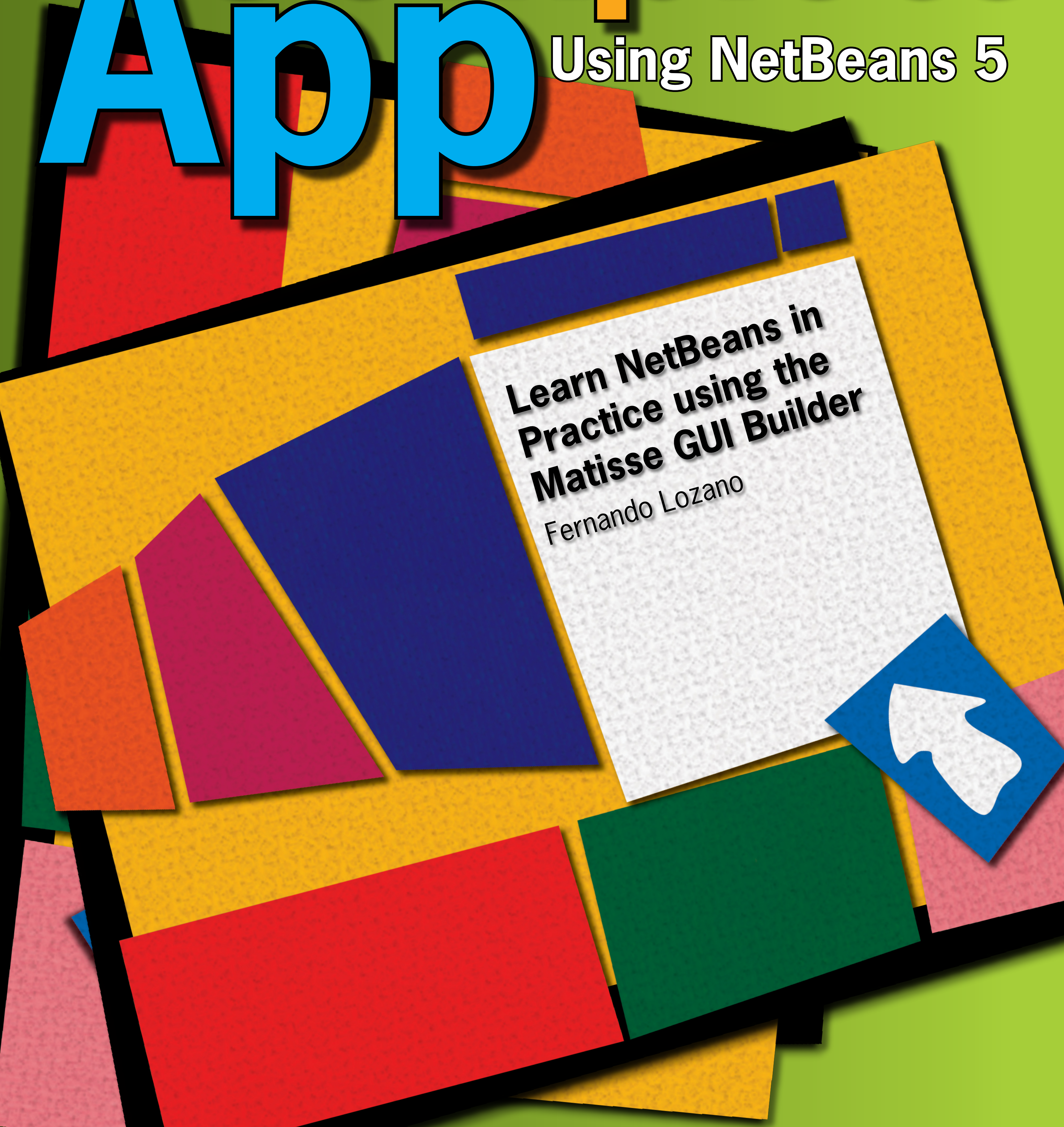
Extending the IDE
Build your own plug-in modules step by step

Bringing Light to Java Development

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.

1

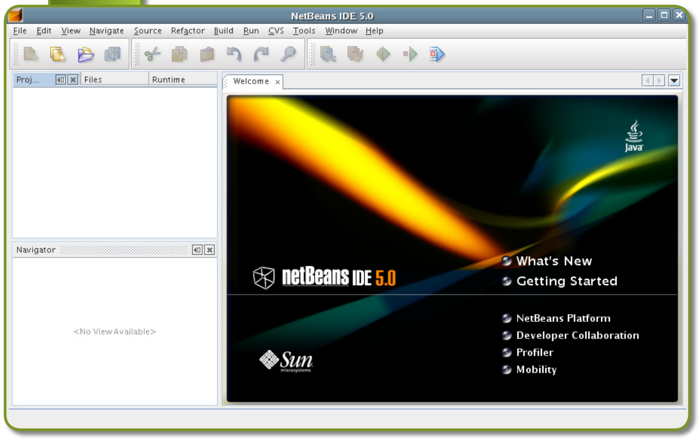
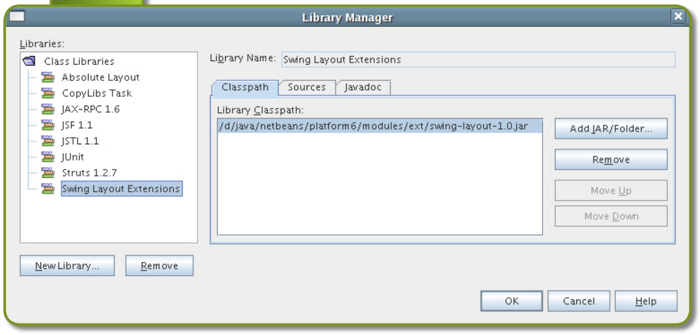


Figure 1 The NetBeans main window and welcome page

2


Library Manager

Libraries:

- Class Libraries
  - Absolute Layout
  - CopyLibs Task
  - JAX-RPC 1.6
  - JSF 1.1
  - JSTL 1.1
  - JUnit
  - Struts 1.2.7
  - Swing Layout Extensions

Library Name: Swing Layout Extensions

Classpath | Sources | Javadoc

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

Add JAR/Folder... | Remove | Move Up | Move Down

New Library... | Remove

OK | Cancel | Help

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

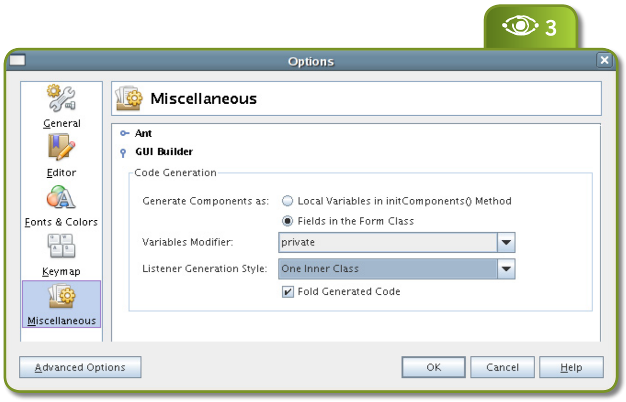
The left area contains two navigational panels. The top one is shared by the Projects, Files and Runtime windows. The bottom area contains the Navigator window, and the right-center area is used for the many editors included with NetBeans. Multiple editors and windows can share the same area; the IDE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

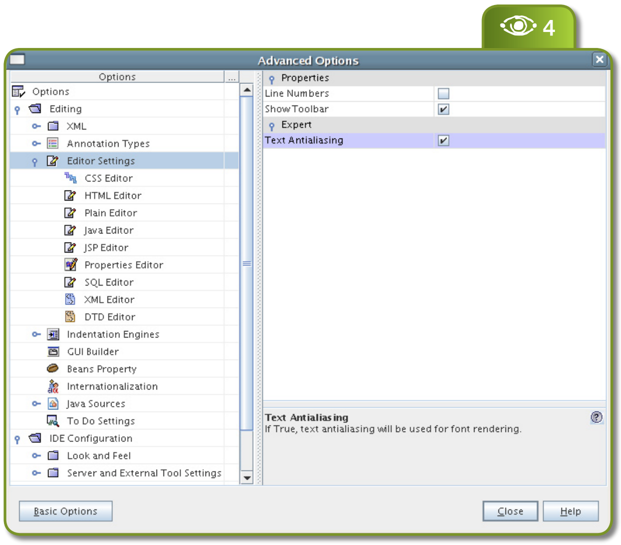
The IDE provides a lot more customization than is shown by the Options window. Just click on the *Advanced Options* button and


Options

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


Advanced Options

```
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

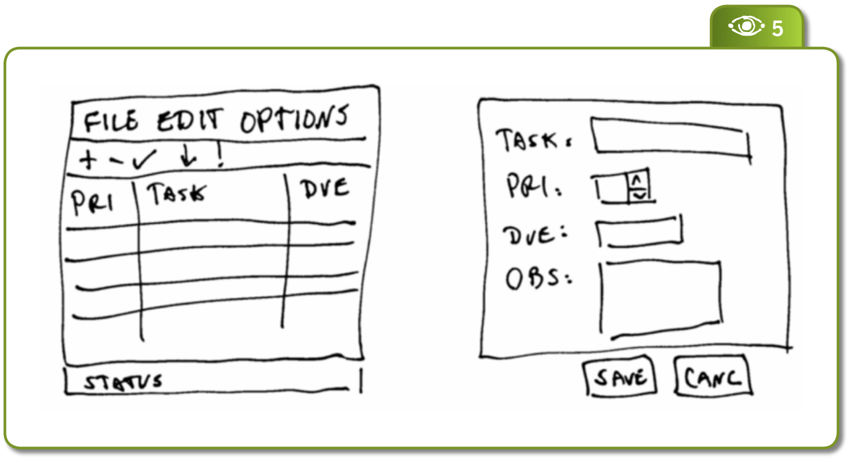


Figure 5
A sketch for the Todo user interface

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

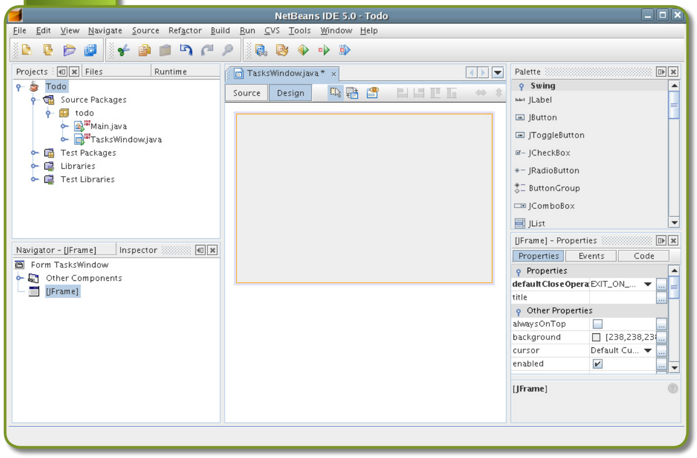


Figure 6 Visual editor with an empty JFrame

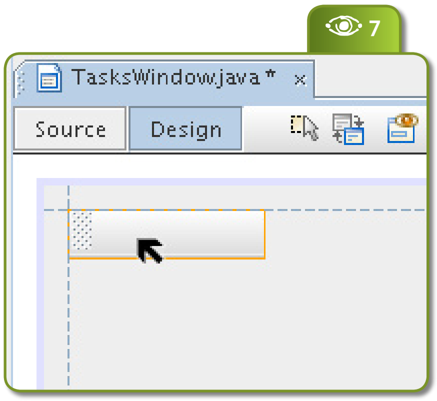


Figure 7
Visual guidelines help positioning and sizing controls in the visual editor

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

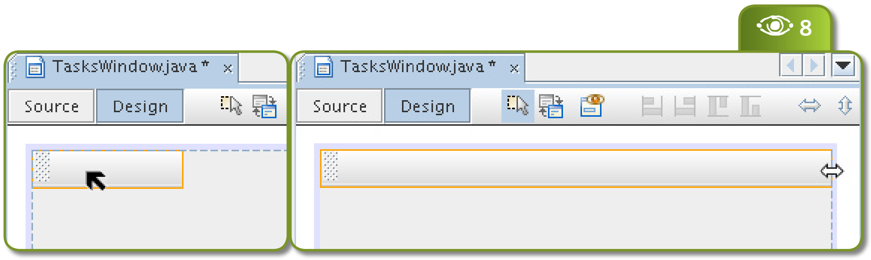


Figure 8
Positioning and resizing the toolbar so it is attached to the left, top and right corners of the JFrame. Notice the guidelines over the drawing borders

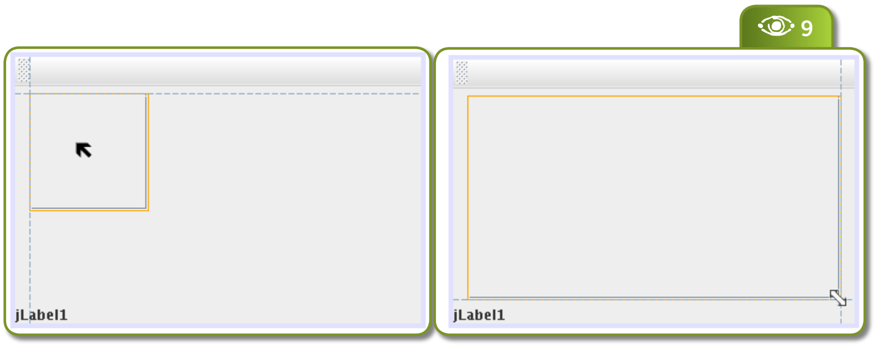


Figure 9
Positioning the JLabel and the JScrollPane

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.

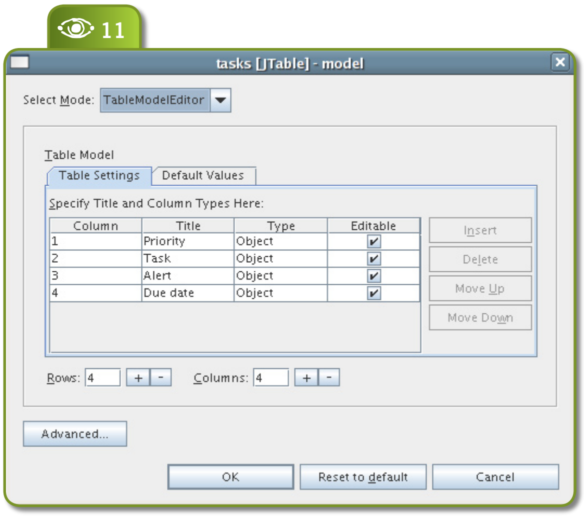
Non-Java files inside the *Source Package* folder will be added by NetBeans to the application jar file, and the application will be able to refer to them as classpath resources, no matter where the jar file is installed on the user machine. As a bonus, the application code doesn't have to worry about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

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


tasks [JTable] - model

Select Mode: TableModelEditor

Table Model

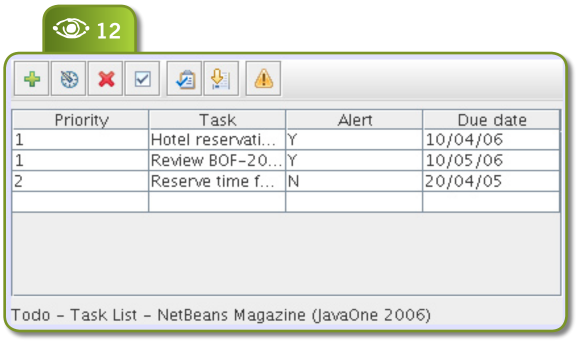
Table Settings | Default Values

Specify Title and Column Types Here:

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

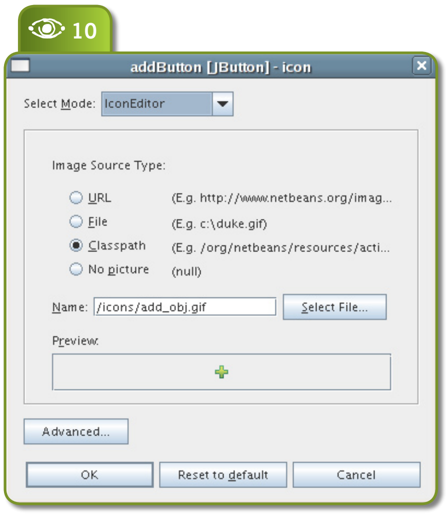
Rows: 4 Columns: 4

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


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

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

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.

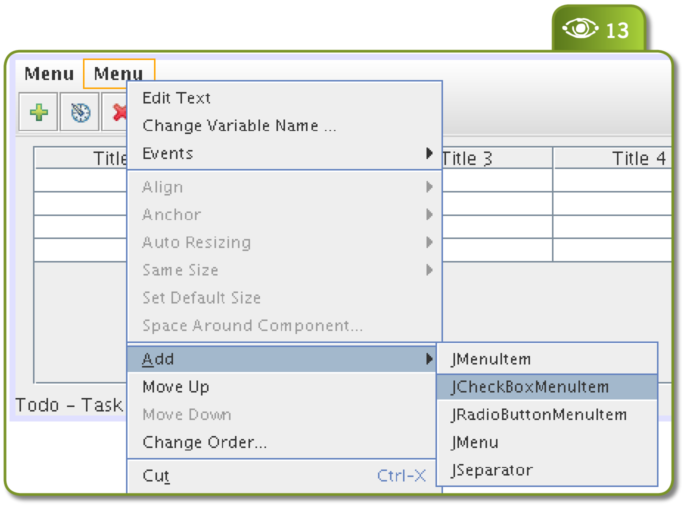


Figure 13
Adding items do a JMenu inside a JMenuBar

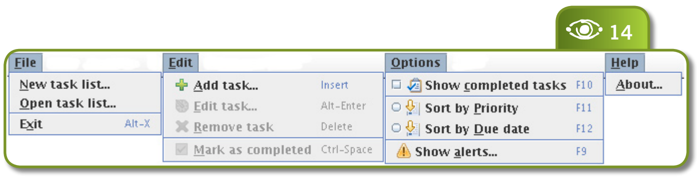


Figure 14
Complete menus for the TasksWindow JFrame

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!

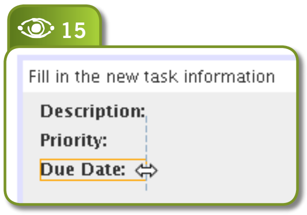


Figure 15 Sizing the TaskDetailsDialog labels

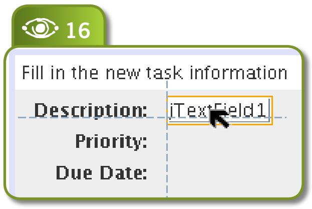


Figure 16 The "free design" layout aligns different kinds of controls by the text baseline

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**.

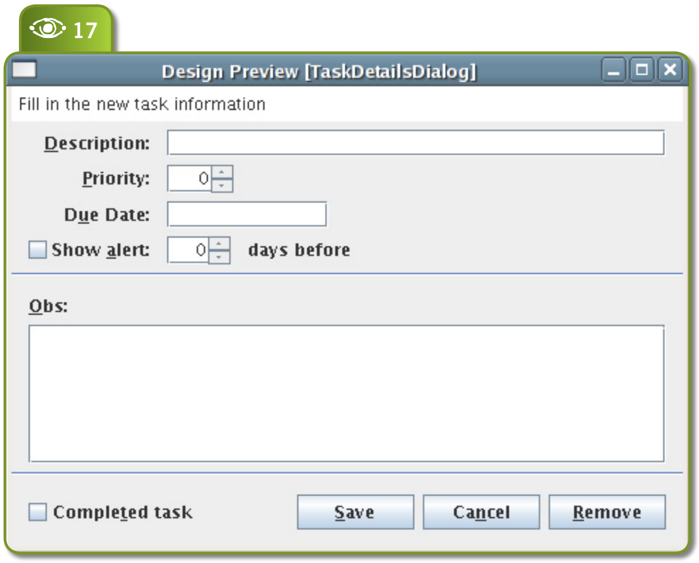


Figure 17 Finished prototype for the TaskDetailsDialog

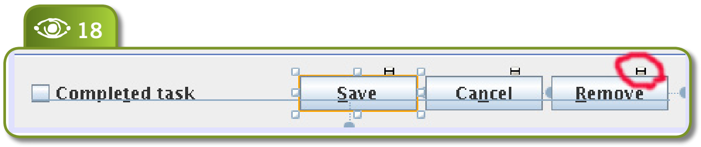
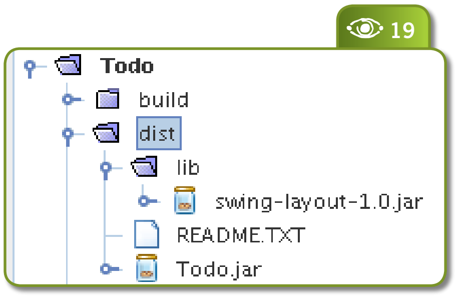


Figure 18 The IDE's visual hint for same-size components

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

Todo
build
dist
lib
swing-layout-1.0.jar
README.TXT
Todo.jar

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.

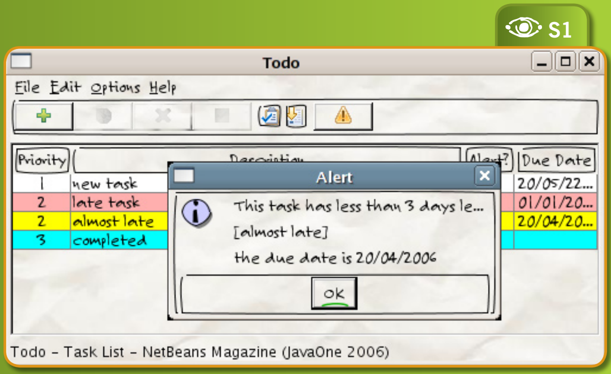
4. Rebuild the application so the *napkinlaf.jar* archive gets copied to the *dist/lib* folder and the jar manifest gets updated with a reference to the new library.

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.


S1

Todo

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		

Alert

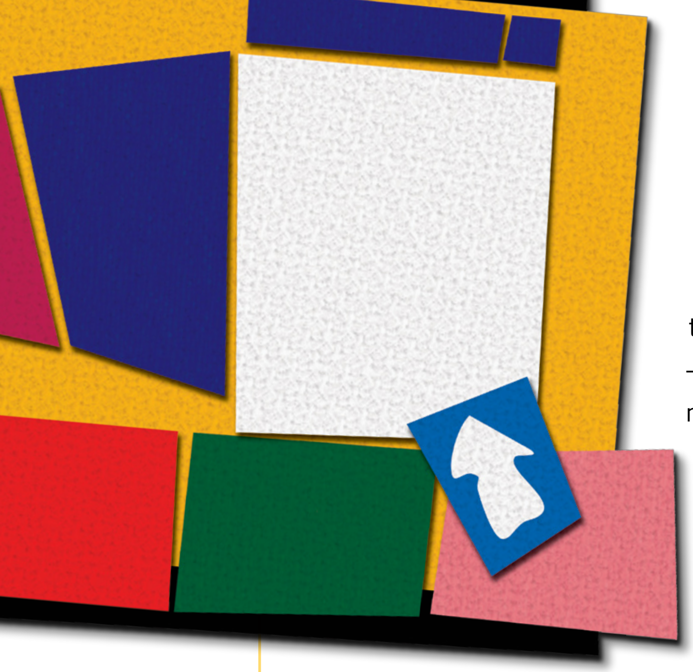
This task has less than 3 days le...
[almost late]
the due date is 20/04/2006

Todo - Task List - NetBeans Magazine (JavaOne 2006)

Figure S1 Sketch for the Todo user interface

napkinlaf.sf.net

Napkin custom Swing LAF home page



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

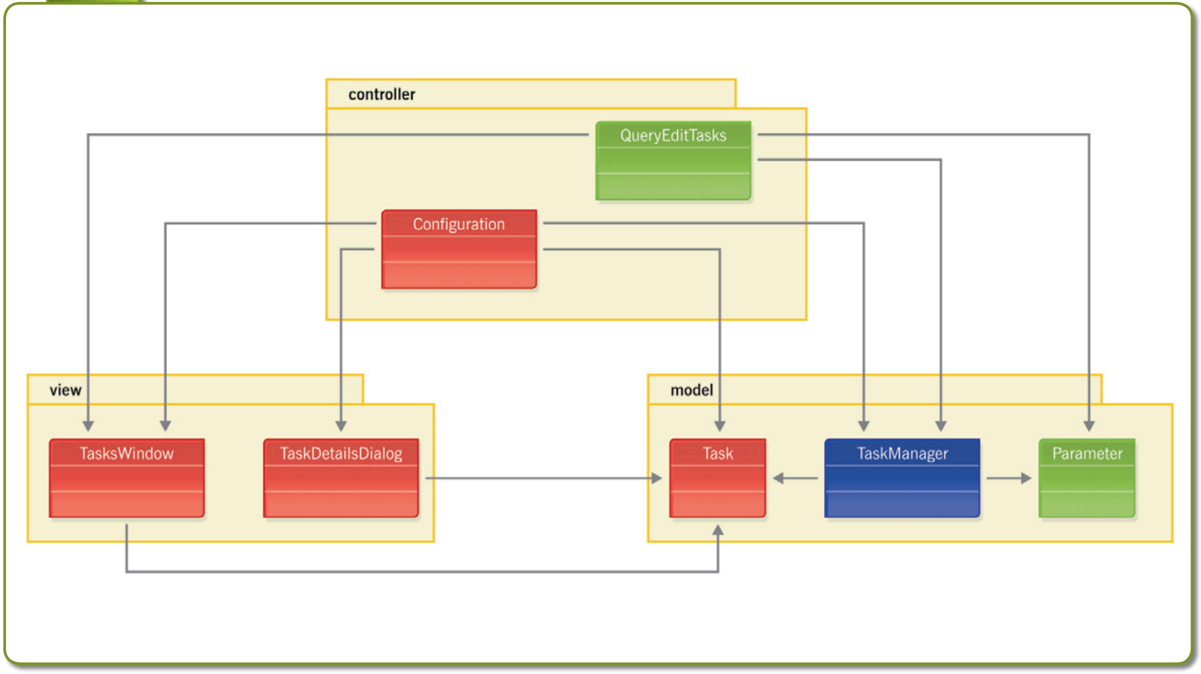
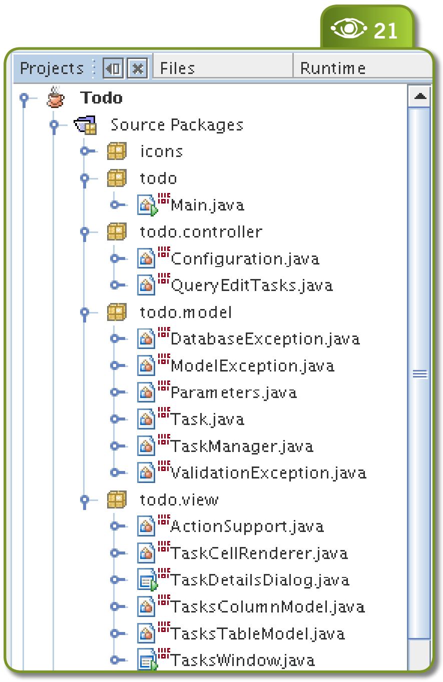


Figure 20 UML class diagram for the main application classes

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.



```
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:

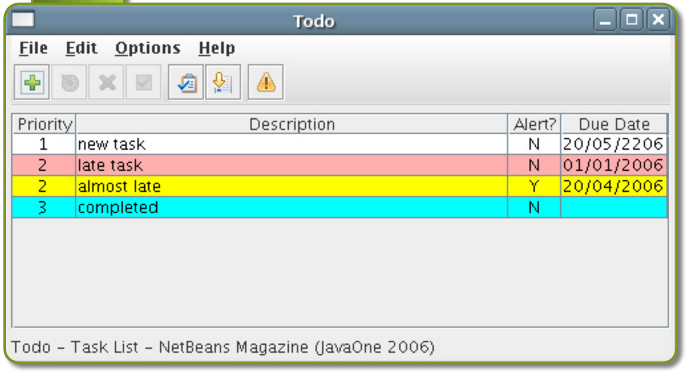
1. Internal events, that affect just the view itself.
2. External events, which cause model methods to execute.

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 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.

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

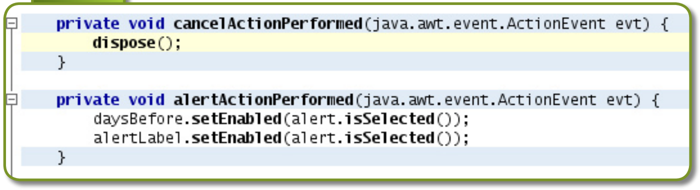
22



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.

Inspecting the Database

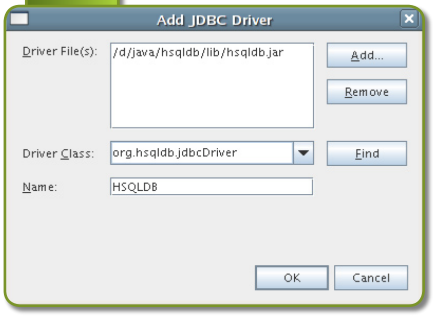
When developing and debugging persistence code, developers usually need a way to tap into the database. Maybe they need to check the effect of an update, or change some table definition. NetBeans provides direct support for browsing any JDBC-compliant database and submit SQL commands.

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.

hsqldb.org

HSQLDB, an Open Source embedded 100%-Java database

24

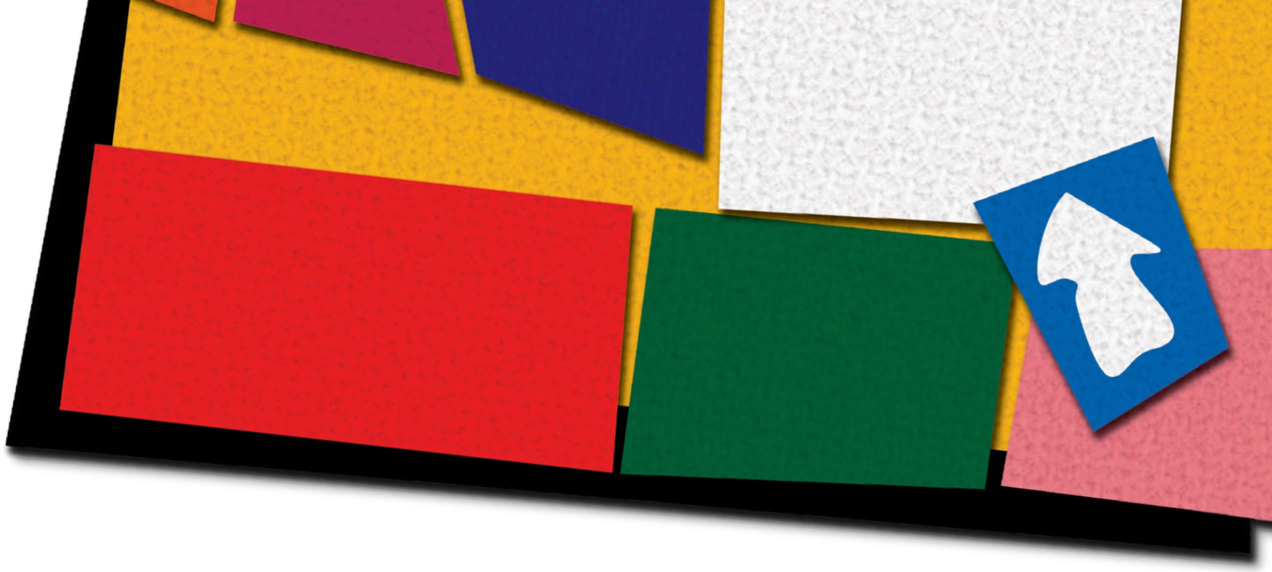

Add JDBC Driver

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.

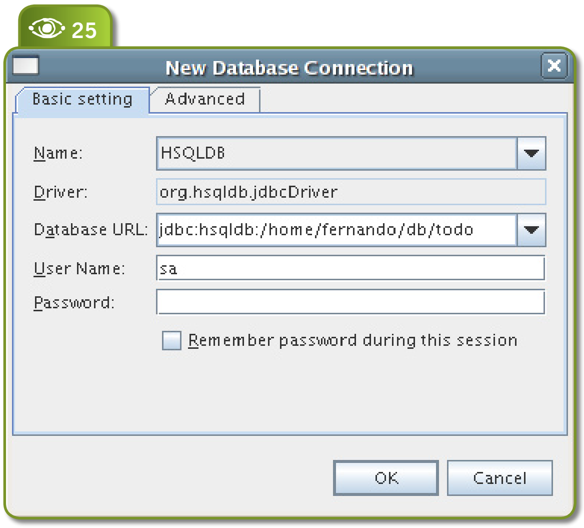
The Todo application uses HSQLDB in the stand-alone mode, which locks the database files for exclusive access. So it won't be possible to use the NetBeans database console while the application is running. However it's possible to run HSQLDB in server mode accepting concurrent connections from multiple 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



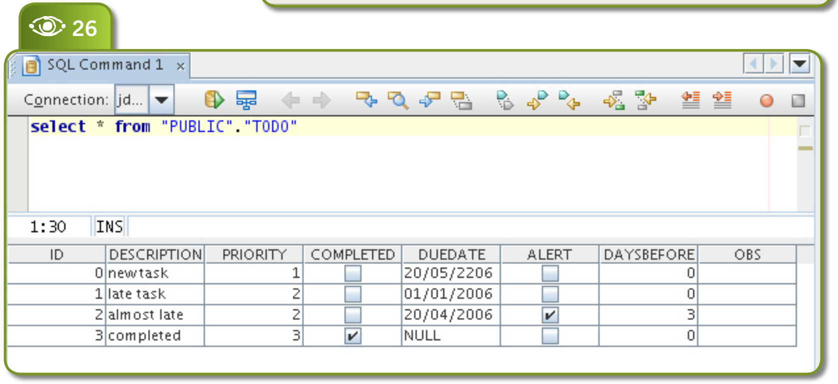
New Database Connection

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

Figure 25 Connecting to the Todo task database



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

1:30 INS

ID	DESCRIPTION	PRIORITY	COMPLETED	DUEDATE	ALERT	DAYSBEFORE	OBS
0	newtask	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	

Figure 26 Executing SQL statements