Modules . Visual Library . Maven 2 . Ruby . Profiling . Web Services . Localization . Schliemann

December . 2007

NetBeans magazine

The Best Ruby IDE
Dynamic web development and the IDE's Ruby features

Fluent in NetBeans
How global localization efforts are expanding the IDE's reach

Advanced Profiling
Real-world explorations with the NetBeans Profiler

Schliemann in the Field
People and projects working to multiply language support

Expand your development Horizons with NetBeans 6.0

Creative uses of the Visual Library
Explore the graph handling and visualization Platform API

Creating RESTful Web Services
A comprehensive tutorial on extensions for REST development

Module Development with Maven
A powerful alternative for building NetBeans extensions

December . 2007

NetBeans magazine

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

Assistant Editor
Osvaldo Doederlein
opinali@gmail.com

Design and Layout
pH Design

Graphic Designers
Tarcísio Bannwart
Jaime Peters Jr.
Tersis Zonato

Illustrator
Felipe Machado

Contributors
Emilian Bold
Fabrizio Giudici
Geertjan Wielenga
Janice Campbell
Masaki Katakai
Osvaldo Doederlein
Peter Liu
Roman Strobl

Community Support
Robert Demmer
Bruno Souza

NetBeans Magazine is supported by NetBeans.org

New Horizons

Release 6.0 is finally here, and the NetBeans Community continues to grow both in the programming and the natural language dimensions. This issue showcases the ever-expanding scope of the IDE and Platform, covering topics from Web 2.0 development to module construction and IDE translations.
One of the main highlights in the roster of new 6.0 features is Ruby development support, which is already attracting major new blood to the NetBeans community. The opening article gives you a thorough overview of the many features already available, which have been making quite an impression among Ruby developers.

However, you don't need to wait for new releases to use other languages in the IDE. There's always the Generic Languages Framework, or Project Schliemann. A new article by Schliemann's most enthusiastic evangelist shows how this once exploratory but now production-quality framework is being used to add language support in areas varying from essential to niche.

As I write, there are NetBeans releases localized to more than 15 languages, and the number is growing. Though English is the modern lingua franca, non-English speaking developers feel much more at home in an IDE in their native language – especially those from developing countries or regions, who in general have less access to foreign-language instruction. An article by one of the leaders of the localization efforts at Sun shows how this is happening, the challenges encountered, and what you can do to help out.

This issue also brings two articles from world experts on the NetBeans Platform. You'll see how the new Visual Library API is being used in the Platform-based blueMarine application as the basis for sophisticated image management components. And another in-depth article covers all you need to know in order to use the IDE, Maven 2 and plugins to build your NetBeans modules – an alternative to the standard Ant-based harness.

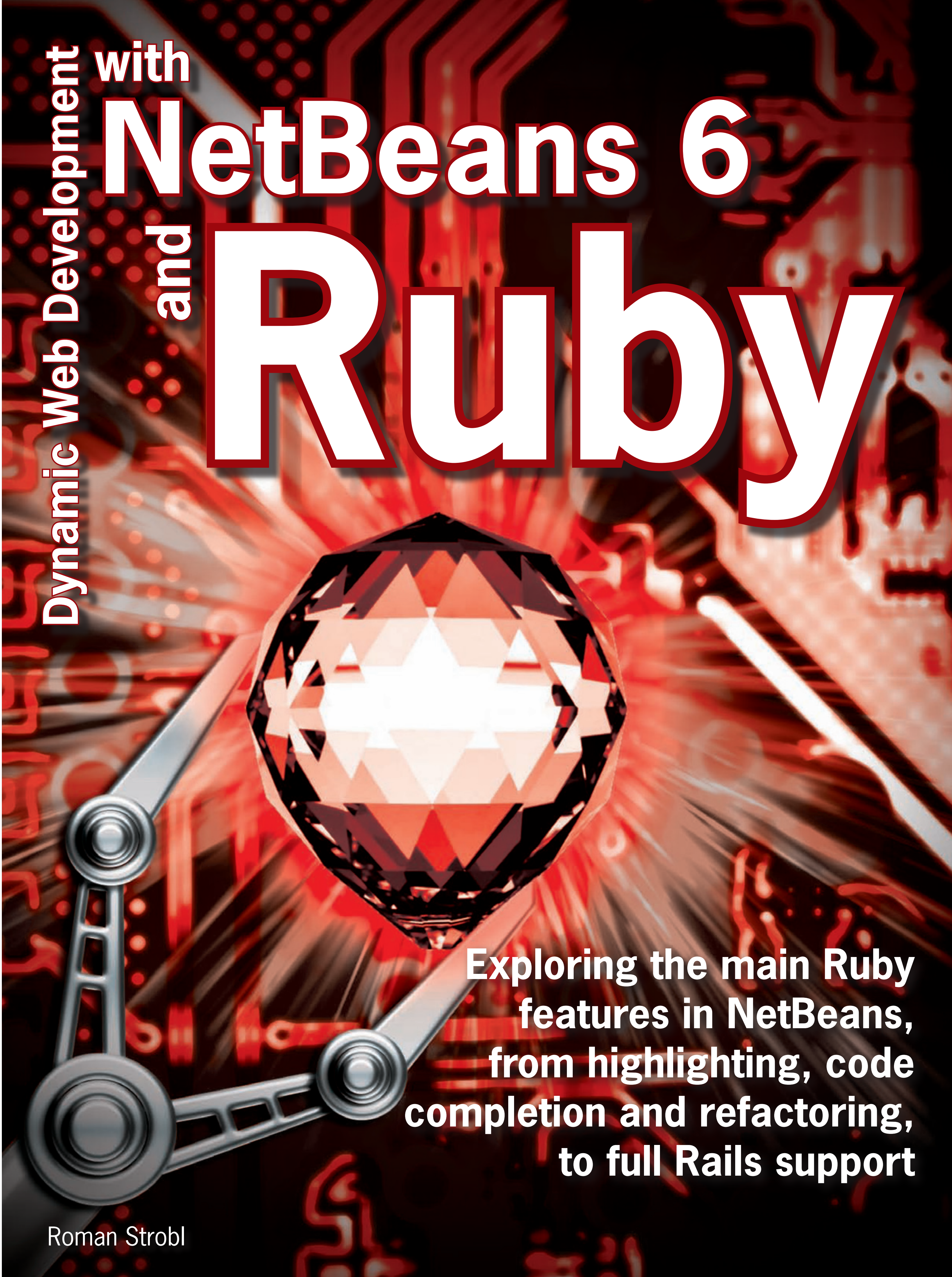
Finally, don't miss the detailed article on profiling. By analyzing aspects of a complex project, the author shows how to use the NetBeans Profiler (including its top new features) to gain insights into the workings of your Java applications. You then see how to use this information to identify optimization opportunities and act on them, while keeping tradeoffs and potential problems in context.

Happy coding!

Leonardo Galvao

Contents

Dynamic Web Development with NetBeans 6 and Ruby



Exploring the main Ruby features in NetBeans, from highlighting, code completion and refactoring, to full Rails support

Roman Strobl

NetBeans has always supported the latest Java standards, and the developer community around the tool and Platform consists largely of Java developers. But as time progresses, a growing number of developers are realizing the advantages of using more than just one language. In some domains, this is actually a must – for example if you want to talk directly to the underlying hardware. NetBeans takes care of that with its C/C++ support.

In some other domains, using a different language is not essential but can bring advantages, such as faster development and quicker turnaround, or better adaptation to requirement changes. There are various scripting languages that have been popular in the web development space for years, like PHP and Perl. Developers who use these love their speed of development and ease of deployment. However, building larger and maintainable web applications with such languages has always been a challenge. You can get things done fast as a single developer, but it's harder to work in a team or take over someone else's code. Indeed, most developers will agree that Java is better for building large-scale, well-designed and maintainable web applications.

Java SE 6 brought interesting changes in this area. From that release on, as you know, scripting languages are officially supported on top of the Java platform. JDK 6 already bundles the Rhino scripting engine for JavaScript, and the JVM is opening up to a variety of other scripting languages. Times are especially exciting for open-minded Java developers, who now have the possibility to mix and match a variety of languages, and can count on tooling support that's quickly improving.

For example, you can use Groovy, a very dynamic language that compiles to Java bytecode; or choose Ruby and run it on top of the JVM thanks to the JRuby runtime. The trend continues with the Java SE 7, which will bring further improvements, like a new *invokedynamic* bytecode (JSR-292) for increasing the performance of dynamic languages compiled into Java bytecode.

These changes in the platform create many new possibilities. You can benefit from the large amount of Java libraries and the JVM itself, while using a dynamic approach to developing web applications. Frameworks such as Ruby on Rails or Grails (Groovy's equivalent to RoR) greatly simplify web development by using a large number of defaults ("convention over configuration"). They also provide powerful code generators, scaffolding, and more. So you can, for example, use Java for the back-end code (where Java really shines) and use a dynamic language for the web front end.

Unlike some of the older dynamic languages, Ruby and Groovy are both very cleanly designed, with object orientation in mind since their birth. And I would argue that they are more suitable for larger applications and for team development, too. Add this to the possibility to use Java APIs easily, and they become very compelling to Java developers. Note that these scripting languages are interesting not only in the web space, where they get most of the interest, but can also be used to script Swing GUIs or glue different applications together.

NetBeans and language support

Java developers spend most of their time in their IDEs and many can't do without the comfort these tools provide. When Java developers want to try a new language or framework, the first thing they will probably do is search for a plugin for their favorite IDE, to simplify the learning process and get productivity from the start. So it's important for NetBeans to support new scripting languages, especially as they become more popular among Java developers.

When discussing language support in NetBeans it's worth talking about Schliemann. This project (new in NetBeans 6.0) allows you to define a programming language and integrate it with the IDE. You can define what parts of the language should be displayed in the navigator and how to indent, fold and highlight code, plus many other features. (You can learn more about the framework in the

wiki.netbeans.org/wiki/view/Ruby
Wiki for Ruby support in NetBeans

languages.netbeans.org
Project Schliemann

Ruby NetBeans homepage
ruby.netbeans.org

JRuby homepage
jruby.codehaus.org

two articles by Geertjan in this and the previous issues of NetBeans Magazine.)

I realize that not everyone is interested in writing language support into NetBeans, so let's look at which languages are currently supported. NetBeans 6.0 provides first-class support for Ruby and the Ruby on Rails (RoR) framework. JavaScript is also supported, with features such as syntax highlighting and code completion. In addition, there's work being done on a PHP plug-in; its first beta version will be published together with the NetBeans 6.0 release.

Work is being done on plug-ins for Groovy, Velocity Template Language, Scala, and several other languages. And more projects are probably being started as you read this text. You can search the plug-in manager and plug-in portal for your favorite language, and if it's not supported, don't forget that NetBeans is an open-source project, and that contributing by writing support for a new language can be an ideal way to give back to the community!

Ruby on Rails – What is the Buzz About?

Ruby was started by Japanese developer Yukihiro "Matz" Matsumoto in 1993, and had its first release in 1995. The language's creator has said that Ruby is designed for programmer productivity and fun, following the principles of good user interface design. But Ruby has not really become widely popular until another guy – David Heinemeier Hansson, "DHH" – created an MVC framework called Rails, which is focused on ease of development.

Rails provides features such as scaffolding, object-relational mapping and code generation. Many other frameworks provide such features as well, but Ruby on Rails is quite unique in making common development tasks as easy to do as possible, and thus making developers more productive. With strong emphasis on configuration by exception, as well as powerful code generators and clean APIs, it can indeed help developers with most mundane tasks. But Rails wouldn't be as good if there was no Ruby behind it. Many lines of code can be saved thanks to Ruby's dynamical nature – generating code on the fly is a commonly used technique in RoR, for example to handle object-relational mapping. Almost every Java framework requires some code duplication, which RoR avoids by letting you configure application parameters in a single place.

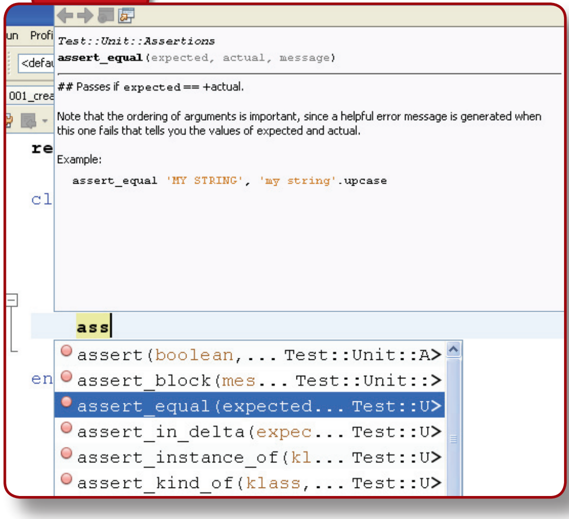
Editing Ruby code in NetBeans

One of the most important features in any IDE is code completion, of course. If you write Java code, you probably can't imagine your life without it. Implementing code completion for Ruby is harder than it may seem, however. The reason is that there are no static types in the Ruby language, so the IDE needs to "guess" them, unlike in Java where types are always known.

NetBeans uses sophisticated heuristics to offer the best options in the code completion window (see **Figure 1**). Code completion helps not only by showing possible methods, classes or modules, but also by providing access to documentation, which is extremely useful if you are learning the language and it's APIs (see **Figure 2**).

Like in Java code, you can see all occurrences of a variable by moving your cursor over it. You can also change these occurrences easily by pressing Ctrl+R (a feature called *Instant Rename*); **Figure 3** shows an example. As you can notice, many of the features of the new Java editor in NetBeans 6.0 are also available for Ruby editing. Syntax highlighting is also very advanced

1



```
Test::Unit::Assertions
assert_equal(expected, actual, message)

## Passes if expected == +actual.

Note that the ordering of arguments is important, since a helpful error message is generated when
this one fails that tells you the values of expected and actual.

Example:
  assert_equal 'MY STRING', 'my string'.upcase

ass
assert(boolean,... Test::Unit::A>
assert_block(mes... Test::Unit::>
assert_equal(expected... Test::U>
assert_in_delta(expec... Test::U>
assert_instance_of(kl... Test::U>
assert_kind_of(klass,... Test::U>
```

Figure 1. Code completion in a Ruby test case

for Ruby, as **Figure 4** shows.

You can use many other features Java developers are accustomed to, such as hyperlinking (*Go to Declaration*), code folding, a navigator, code templates, etc. But what will surprise even the biggest skeptics is the possibility to refactor Ruby code (see **Figure 5**). You can search for usages and use the *Rename* refactoring, and other refactorings are currently in the works.

Ruby on Rails Support

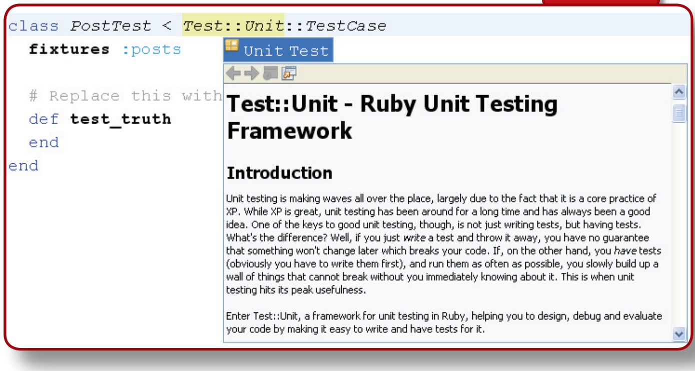
NetBeans 6.0 includes first-class support for Ruby on Rails through special project templates, as you can see in **Figure 6**. Traditionally Rails development was done from the command line: developers launched different commands – called generators – which created different folders and files. NetBeans integrates all the Rails generators into different wizards, so you don't have to go outside the IDE to execute them (though you can still do that if you prefer). See one of these wizards in action in **Figure 7**.

Once you're done with development of the Rails application, you can simply hit the *Run* button: the Webrick server starts up and your web application opens in a web browser. (Webrick is a simple Ruby library that provides web server services.) The turnaround speed is pretty fast with Ruby, since the build and deploy steps are omitted. You can do any updates in the IDE and just refresh the web browser to see the changes in the application.

The Webrick server is intended mostly for development (it's written in Ruby), however it's easy to switch, for example, to Mongrel, a production-quality server that can be used to host the final web application.

Debugging

The debugging experience in Ruby is very similar to that using the Java debugger. You can set breakpoints, browse the call stack, add watches, evaluate expressions, and more. When running the code in the debugger you can step into, step out, step over… So



```
class PostTest < Test::Unit::TestCase
  fixtures :posts

  # Replace this with
  def test_truth
  end
end
```

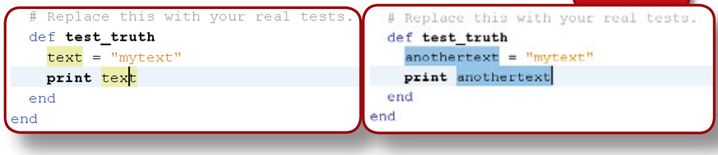
Test::Unit - Ruby Unit Testing Framework

Introduction

Unit testing is making waves all over the place, largely due to the fact that it is a core practice of XP. While XP is great, unit testing has been around for a long time and has always been a good idea. One of the keys to good unit testing, though, is not just writing tests, but having tests. What's the difference? Well, if you just *write* a test and throw it away, you have no guarantee that something won't change later which breaks your code. If, on the other hand, you *have* tests (obviously you have to write them first), and run them as often as possible, you slowly build up a wall of things that cannot break without you immediately knowing about it. This is when unit testing hits its peak usefulness.

Enter Test::Unit, a framework for unit testing in Ruby, helping you to design, debug and evaluate your code by making it easy to write and have tests for it.

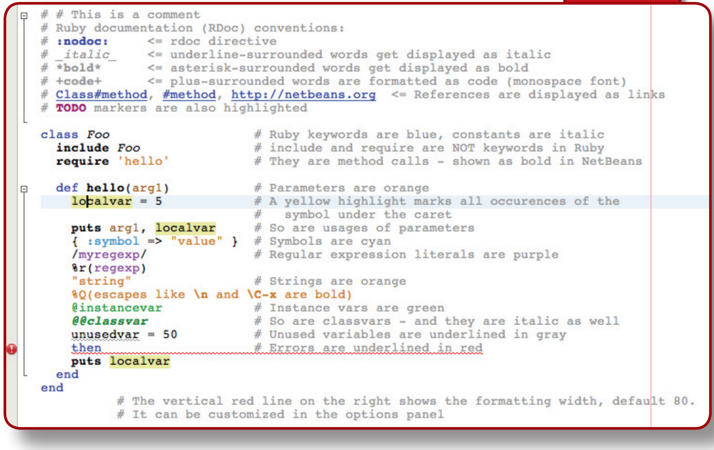
Figure 2. Viewing documentation while editing in a Ruby class



```
  # Replace this with your real tests.
  def test_truth
    text = "mytext"
    print text
  end
end
```

```
  # Replace this with your real tests.
  def test_truth
    anothertext = "mytext"
    print anothertext
  end
end
```

Figure 3. Instant renaming

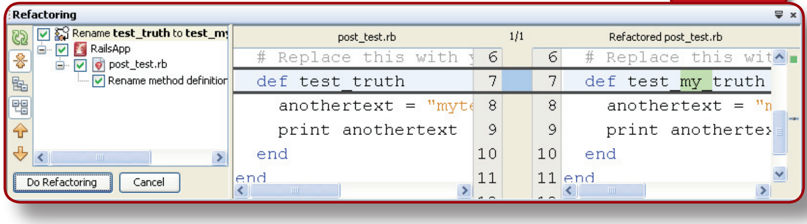


```
# # This is a comment
# Ruby documentation (RDoc) conventions:
# :nodoc:     <= rdoc directive
# _italic_    <= underline-surrounded words get displayed as italic
# *bold*      <= asterisk-surrounded words get displayed as bold
# +code+      <= plus-surrounded words are formatted as code (monospace font)
# Class#method, #method, http://netbeans.org  <= References are displayed as links
# TODO markers are also highlighted

class Foo                      # Ruby keywords are blue, constants are italic
  include Foo                  # include and require are NOT keywords in Ruby
  require 'hello'              # They are method calls - shown as bold in NetBeans

  def hello(arg1)              # Parameters are orange
    localvar = 5               # A yellow highlight marks all occurences of the
                               #   symbol under the caret
    puts arg1, localvar        # So are usages of parameters
    { :symbol => "value" }     # Symbols are cyan
    /myregexp/                 # Regular expression literals are purple
    %r(regexp)
    "string"                   # Strings are orange
    %Q(escapes like \n and \C-x are bold)
    @instancevar               # Instance vars are green
    @@classvar                 # So are classvars - and they are italic as well
    unusedvar = 50             # Unused variables are underlined in gray
    then                       # Errors are underlined in red
    puts localvar
  end
end
          # The vertical red line on the right shows the formatting width, default 80.
          # It can be customized in the options panel
```

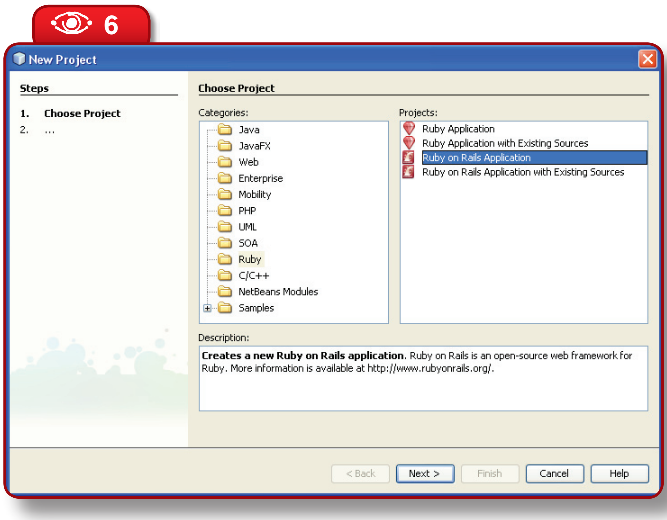
Figure 4. Ruby syntax highlighting



```
  # Replace this with y
  def test_truth
    anothertext = "myte
    print anothertext
  end
end
```

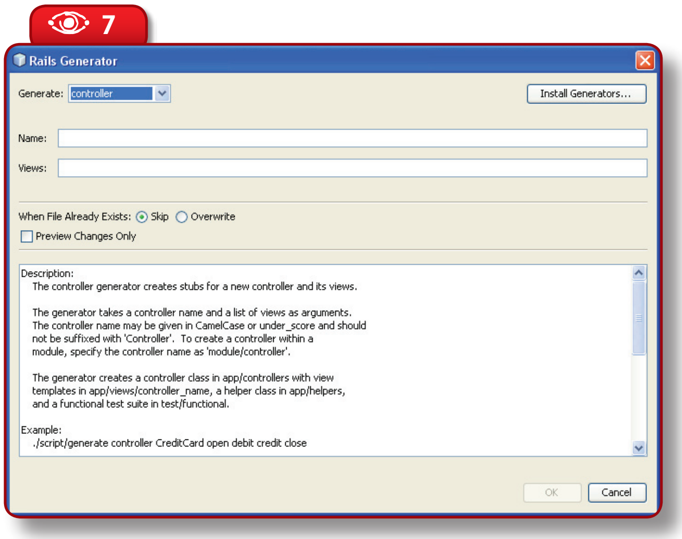
```
  # Replace this wit
  def test_my_truth
    anothertext = "n
    print anothertex
  end
end
```

Figure 5. Comparing results before refactoring



Creates a new Ruby on Rails application. Ruby on Rails is an open-source web framework for Ruby. More information is available at http://www.rubyonrails.org/.

Figure 6. Selecting Ruby and Rails projects in the New Project wizard



```
Description:
   The controller generator creates stubs for a new controller and its views.

   The generator takes a controller name and a list of views as arguments.
   The controller name may be given in CamelCase or under_score and should
   not be suffixed with 'Controller'.  To create a controller within a
   module, specify the controller name as 'module/controller'.

   The generator creates a controller class in app/controllers with view
   templates in app/views/controller_name, a helper class in app/helpers,
   and a functional test suite in test/functional.

Example:
   ./script/generate controller CreditCard open debit credit close
```

Figure 7. Wizard-based Rails code generation

if you've used a graphical debugger before, you'll know how to use the Ruby debugger. You can also debug RHTML files (RHTML is a dynamic page system much like JSP 1.0), as seen in **Figure 8**.

Unit testing

Unit testing support is very well integrated too. For instance, when you create a new model, a unit test is automatically generated for you. This is a feature of Ruby on Rails; its philosophy is that testing is so important that it should not be omitted.

You can also run tests right from the editor and navigate to test classes. If you are really into testing, you can install the "ZenTest" Ruby Gem (use *Tools|Ruby Gems*). This adds an *AutoTest* item to your project's context menu. When invoked, it will launch AutoTest on your project, which will run unit tests automatically whenever you modify a file. In many cases, AutoTest can figure out which unit tests need to be run – this is especially true for Rails projects. If not, it will run all unit tests.

blogs.sun.com/tor

Tor Norbye's blog

Ruby hints

Ruby hints (also known as quick fixes) are indicated as light bulbs at the side of the editor window and help you resolve issues with your code. They can show that some code might have unintentional side effects, or tell you that a deprecated API is being used. Another Ruby hint helps you reformat a long line of code into multiple lines.

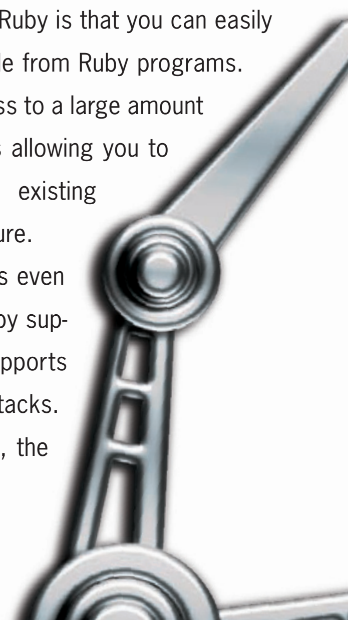
Many additional hints are in development, being available through the plug-in manager. Their development was started after feature freeze so they couldn't be part of the official release. They are very worth a look and will make your Ruby development even more comfortable.

JRuby or native Ruby?

NetBeans comes with the JRuby runtime, which is used by default for all Ruby-related tasks. JRuby, an implementation of Ruby written in Java, can both interpret Ruby sources and compile them into Java bytecode that will run efficiently on a JVM. You can easily switch to the native Ruby interpreter if you prefer to use it (e.g. for performance reasons). Just change the Ruby binary in *Tools|Options>Ruby* to your native Ruby interpreter.

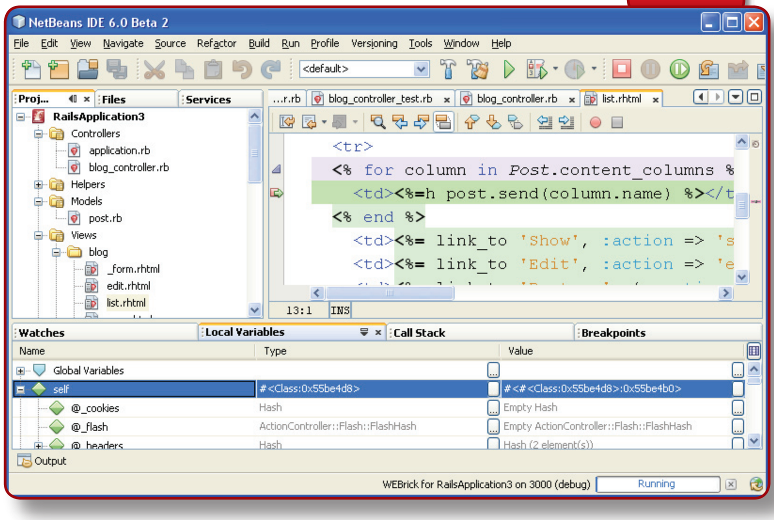
The biggest advantage of using JRuby instead of "pure" Ruby is that you can easily invoke Java code from Ruby programs. This gives access to a large amount of Java libraries allowing you to leverage your existing Java infrastructure.

NetBeans goes even further in its Ruby support. It fully supports two full Ruby stacks. The first choice, the



standard toolchain used by traditional Ruby and RoR programming, includes other items like the gems package manager (accessible through a wizard in NetBeans), the Webrick and Mongrel web servers, and the Rake build manager.

In the second alternative, a Java platform-oriented stack, the JRuby runtime is complemented by JDBC support (a boon because JDBC drivers are superior and available for many more products than Ruby's own database drivers). There's also support for deployment in Java EE servers like GlassFish V2.



```
<tr>
<% for column in Post.content_columns %
  <td><%=h post.send(column.name) %></t
<% end %>
  <td><%= link_to 'Show', :action => 's
  <td><%= link_to 'Edit', :action => 'e
```

Name	Type	Value
Global Variables		
self	#<Class:0x55be4d8>	#<#<Class:0x55be4d8>:0x55be4b0>
@_cookies	Hash	Empty Hash
@_flash	ActionController::Flash::FlashHash	Empty ActionController::Flash::FlashHash
@_headers	Hash	Hash (2 element(s))

WEBrick for RailsApplication3 on 3000 (debug) Running

Figure 8. RHTML debugging

Additional features and plug-ins

There are many other features available which were not discussed in this article. You can create your own live templates to speed up coding and there is a spell checker for RubyDoc available from the plugin manager. Also, as Ruby developers are very editor-centric, we provide a set of different plug-ins focused on a variety of editing tasks, such as rectangular selection, highlighting of trailing spaces and tabs, quick file choosing, etc. You can find out more about additional plug-ins at *wiki.netbeans.org/wiki/view/RubyPlugins*.

Conclusions

This article introduced various features for Ruby development available in NetBeans 6. Many Ruby developers are very excited about this new functionality. Indeed, you'll find quite a few blog entries from developers saying they've switched to NetBeans just because of its Ruby support.

I believe this is just the beginning of the "dynamic language journey" for NetBeans. Large parts of the code in the Ruby feature set are language independent, thus in future NetBeans releases we can expect better tooling for Groovy, PHP, Python and other popular languages. NetBeans is on its way to becoming a very useful tool for more and more developers, regardless of which programming language they choose.

Roman Strobl (*roman.strobl@sun.com*) works at Sun Microsystems as technology evangelist with focus on development tools. He has many years of experience doing software development in Java and all sorts of dynamic languages, and is a frequent speaker at Java conferences and Java User Group meetings around the world. He is also a passionate blogger and producer of the NetBeans Podcast. Roman is co-founder of the Czech Java User Group and enjoys working with open source communities. He is always available to discuss NetBeans, Java, open source and related topics over a glass of beer.

Advanced Profiling

Theory in Practice with NetBeans

Explore the latest Profiler features of NetBeans 6.0 and review profiling techniques and best practices in a real-world tutorial

Osvaldo Pinali Doederlein

In this article, I'll walk you through a series of best practices in Java code profiling, using NetBeans 6.0. The approach is to show a realistic session of profiler-driven code optimization, using the NetBeans Profiler as the core tool.

Hunting for a real-world example

Too many articles, including most of my own production, are limited by sample projects that must "fit". But using real-world code adds a whole new dimension of meaning – and readers don't need to trust the author when he concludes that techniques shown through a contrived example will be effective in real applications.

So I picked a real project: iText, a popular open-source Java library for PDF generation and manipulation. Many developers will be familiar with it or use it indirectly (for example, iText is a dependency of many Java report generators and other tools that output PDF). Also, the work performed by iText is both complex and CPU-bound, so it's a good profiling test bed.

My principle here is to use profiling techniques to learn something interesting about a complex project. And if we're lucky, find performance problems and investigate their solution, with the help of these techniques and in particular of the NetBeans Profiler. Moreover, I selected a project that I knew only as a user, but whose source code I've never read before and an internal architecture I'm not familiar with. Thus I'll not be in advantage over the reader, and we can learn together.

The right tool for the right job

The NetBeans Profiler is not the ideal tool for every optimization-related task. This is not due to any limitations, but because profilers are best for fine-grained or low-level performance investigation. Imagine a Java EE reporting application that's not scaling well. The bottleneck could be some inefficient algorithm, but it can also be due to bad application server tuning, inefficient SQL queries, excessive use of remote calls, and many other factors. At this stage of the investigation, you'll often prefer different tools. For example, GlassFish offers detailed monitoring capabilities, and most database servers provide tools to spot heavy queries. A profiling tool *can* still help here though; the NetBeans Profiler integrates well with application servers. But in my experience this integration is more suited to profiling sections of Java EE code that are hard to test outside the container.

Once you've narrowed the bottleneck to a specific subsystem, at the very least to your own application code or, hopefully, something more specific like "the front-end report generation code", then it's the time to start using a code profiler.

Identifying a problem – and benchmarking it

If you plan to follow this article's step by step, you need to first set up your environment as explained in the **box** "Installation and setup". In the project, open the class *examples/com.lowagie.examples.objects.tables.AddBigTable*. This is one of iText's standard samples and was selected because it's complex enough to be interesting for our purposes. Indeed, this sample creates a ~120Kb, 26-page PDF file filled with a very big table. Not your average HelloWorld sample!

A good profiling session should be like any good scientific experiment. You must isolate the code you want to test from other code, and also avoid interference from environment "noise". This sample is already well isolated. It does nothing beyond creating the PDF file; also there's no alien code involved: no database access, no middleware. There is a potential source of environment noise, however: file I/O. The sample program writes the document to a PDF file, but this I/O operation is not relevant to what we are measuring. A larger application that uses iText would probably compose the PDF to a memory stream and serve it to some GUI, perhaps store it in a cache; but most likely it wouldn't write it to a disk file.

profiler.netbeans.org More information about the NetBeans Profiler

www.lowagie.com/iText iText, the PDF manipulation library that's the victim of this article's experiments

The first step towards our profiling goal, then, is to get rid of the file I/O. This is easy to accomplish. I changed the **AddBigTable** class, replacing "**new FileOutputStream(...)**" with "**new ByteArrayOutputStream(...)**". Notice that the memory stream is big enough to hold the entire finished PDF content, so there won't be reallocations of this buffer to spoil benchmarking precision. Check out the resulting code in **Listing 1** (with changed parts in bold); **Figure 1** shows the project loaded in NetBeans.

Installation and setup

You can follow the steps in this article with any release of NetBeans 6.0 that supports Java development. The Profiler is a built-in feature. Start by creating a project using *New Project>Java>Java Class Library*. Then visit iText's download page at SourceForge (*sf.net/project/showfiles.php?group_id=15255*) and from the *iText* group get the *itext-src-2.0.6.tar.gz* file (or the equivalent zip). Unpack it and copy the top-level *com* directory into your NetBeans project's *src* folder. At this point some classes will report compilation errors due to missing dependencies. To solve that, in the same download page go to the *extrajars* group and download the files *bcmail-jdk14-137.jar* and *bcprov-jdk14-137.jar*. Add those to the project's libraries, and NetBeans should compile the whole project without errors.

Now we just need to add the sample code. Again in the download page, go to the *tutorial* group and get the file *tutorial.tar.gz*. Unpack it and copy the folder examples to the NetBeans project root. In the project's Properties (*Sources/Source Package Folders*), click *Add Folder* and select the *examples* folder.

After the rebuild, you'll notice that four packages contain compilation errors; these are examples that use other dependencies, like JFreeChart or the Servlet API. We're not going to use this part, so you can just delete the four packages, and we're ready to go.

Listing 1. Initial updates to the AddBigTable class

```
public class AddBigTable {
  public static void main (String[] args) {
    // step1
    Document document = new Document(
        PageSize.A4.rotate(), 10, 10, 10, 10);
    try {
      // step2
      PdfWriter writer = PdfWriter.getInstance(
         document, new ByteArrayOutputStream(120000));
      // step3
      document.open();
      // step4
      // ... omitted for space: build the document
    } catch (Exception de) {
      de.printStackTrace();
    }
    // step5
    document.close();
}
```

[1] **Profiling** is the investigation of a program's runtime behavior, specifically for this article, *performance* behavior, i.e. finding out how much computing resources are used and breaking up this use into specific regions or activities of the program. **Benchmarking** is assessing a program's performance against some baseline. In application projects, you might create benchmarks for regression tests: if today's build is 10% slower than yesterday's build, that's a performance regression. Profiling and benchmarking share some common issues, like avoiding dead code. The major technical difference is that benchmarks run at full speed, while profiling typically impacts the performance of the measured program. That's because profiling must report fine-grained information about the subject, and this requires significant resources.

Benchmarking

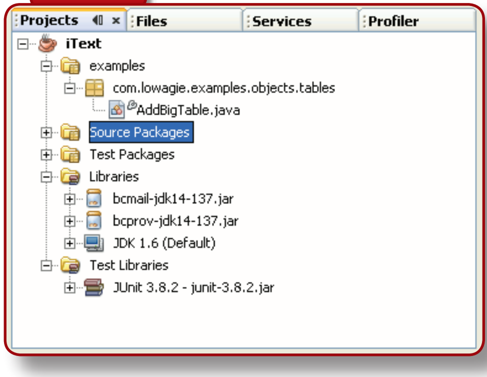
Profiling and benchmarking are of course related disciplines, so before continuing it's interesting to go through an exercise in benchmarking[1]. The basic requirement is avoiding the "Two Deadly Sins of Java Benchmarking": dead code and short runs.

Dead code happens when a benchmark is so simple that smart optimizers notice some code is computing data which is never used – so they just eliminate that code and execution times drop to the floor. This is not the case with our current code, however. Even though I don't use the produced **ByteArrayOutputStream** for any purpose, I know that the path from iText's core through multiple layers of I/O objects (like its **PdfWriter**), and into the byte array stream, is very complex – complex enough that the optimizer in the JVM won't kill it.

However, the program is still guilty of **short run**: short programs will run mostly in interpreted mode, so the measured results will be meaningless. From the command line, I measured 1.890 seconds with HotSpot Client 6.0u3 on a Pentium-IV 2,4GHz, which is not too bad. Ideally though, total execution time would be in the range of tens of seconds to a few minutes, to allow for JVM warm-up (initialization, classloading and dynamic optimization).

This is easy to fix: just add a loop that repeats the test several times. I renamed

1



Projects | Files | Services | Profiler

```
iText
  examples
    com.lowagie.examples.objects.tables
      AddBigTable.java
  Source Packages
  Test Packages
  Libraries
    bcmail-jdk14-137.jar
    bcprov-jdk14-137.jar
    JDK 1.6 (Default)
  Test Libraries
    JUnit 3.8.2 - junit-3.8.2.jar
```

Figure 1. The iText project in NetBeans

the original **main()** method to **test()**, and added a new **main()** that takes the repeat number from the arguments. See the new method in **Listing 2**.

To set up and run this benchmark, go to the project's *Properties>Run>Main Class*, click *Browse* and select our **AddBigTable** class. In the same page make *Arguments=10*. Confirm changes and run the project. I did that for a few recent Java virtual machines; **Table 1** shows the results. For CPU-bound work, the increasing sophistication of JVM technology continues to extract more and more from the same hardware. In this particular test, HotSpot Server 6.0[2] saves 20% of your CPU, compared to release 1.4.2 of the same JVM, or 10% when compared to 5.0.

Profiling CPU usage

We're a couple pages into the article and yet no use of the NetBeans Profiler. This is an important best practice though: planning before profiling. Now we are at the point where powerful tools should really help (and not just entertain us with beautiful data visualizations).

Booting the Profiler is easy. Just run *Profile>Profile Main Project* and sit comfortably while watching the results come in… That, however, is not a very productive approach. I've spent too much time doing exactly this and waiting for the right moment to click "*Reset data*", "*Take snapshot*" or similar buttons in several profilers. What we really need is data from an "area of interest", and to have it discretized per iteration of the test, not accumulated for multiple iterations (the first iterations often have meaningless "cold start" performance). That's why my top new profiling feature in NetBeans 6.0 is **Profiling Points**.

In *AddBigTable.java*, go to the **test()** method's first line, right click it and choose *Profiling>Insert Profiling Point*. Select the *Reset Results* type, accept the defaults for other options and finish

Table 1.

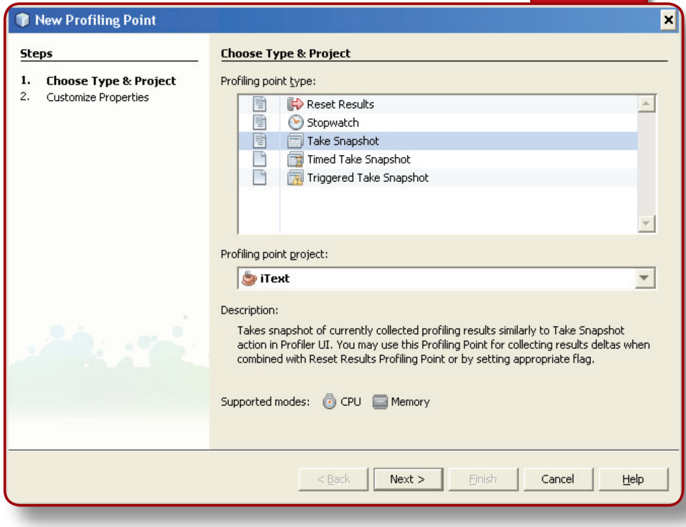
Runtime	Performance
HotSpot Client 1.4.2u16	843ms
HotSpot Server 1.4.2u16	703ms
HotSpot Client 5.0u13	750ms
HotSpot Server 5.0u13	625ms
HotSpot Client 6.0u3	672ms
HotSpot Server 6.0u3	562ms

Table 1. iText's benchmark on Sun JVMs from 1.4.2 up to 6.0; results are best of 10 cycles of each test

Listing 2. New method main() for AddBigTable

```
public static void main (String[] args) {
  int times = Integer.parseInt(args[0]);
  for (int i = 0; i < times; ++i) {
    long t = System.currentTimeMillis();
    new AddBigTable().test();
    t = System.currentTimeMillis() - t;
    System.out.println(t);
  }
}
```

2



New Profiling Point

Steps
1. Choose Type & Project
2. Customize Properties

Choose Type & Project

Profiling point type:
- Reset Results
- Stopwatch
- Take Snapshot
- Timed Take Snapshot
- Triggered Take Snapshot

Profiling point project: iText

Description:
Takes snapshot of currently collected profiling results similarly to Take Snapshot action in Profiler UI. You may use this Profiling Point for collecting results deltas when combined with Reset Results Profiling Point or by setting appropriate flag.

Supported modes: CPU Memory

Figure 2. Creating a Profiling Point

[2] To test with HotSpot Server, set VM Options to --server in the project properties' Run page.

the wizard. Then go to the last line of the same method, and create a *Take Snapshot* profiling point. **Figure 2** shows the first page of the Profiling Point creation wizard.

Having created the two profiling points, we'll collect a snapshot of each run of the **test()** method. Since the **AddBigTable** program loops, invoking this method several times, the *Reset Results* profiling point is important to avoid accumulation of an execution's performance data with that for the previous one. There are other very useful types, like *Stopwatch* (which prints the execution time of a code section) and *Timed Take Snapshot* (for taking a snapshot at a fixed time after reaching the profiling point).

Now start the profiler with *Profile>Profile Main Project*. Select *CPU* and make sure the *Entire Application* and *Use defined Profiling Points* options are checked; also set the *Filter* option to *Profile only project classes* (see **Figure 3**).

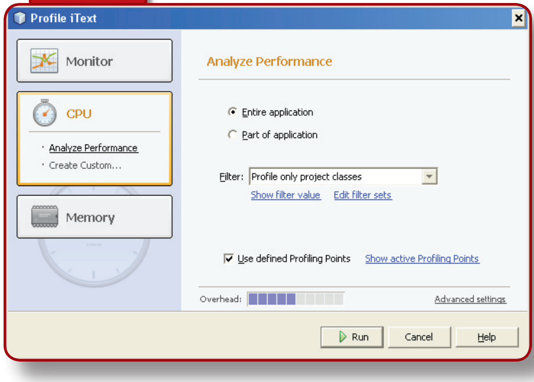
With these settings we instruct the Profiler to observe code execution performance only for classes from our project. As profiling has a significant probe effect (it slows down the observed program), the Profiler allows you to minimize this through several filtering options. The *Advanced settings* dialog also lets you fine-tune profiling precision. For this experiment, however, the default settings will do, and you can click *Run* to start the profiling session.

The result of this session should be similar to **Figure 4**, with ten new snapshots collected. Just look at the last snapshot, which is already "good" (fully optimized execution). You can inspect the results in the *Call Tree* page (top-down view), in the *Hot Spots* page (bottom-up view), or in the *Combined* page, which is split between these two views.

The **ByteBuffer.formatDouble()** method's "hot spot" can be easily identified. Apparently, this method alone takes ~20% of all the running time of the test, which is surely too long for a single method in a project as complex as iText. So, like vultures over rotting meat, let's dive into the code, hoping to find something that can be optimized.

Double click the **formatDouble()** method in the *Call Tree* page to show its source code (see **Listing 3**). This method's purpose is no rocket science: it just formats a

3



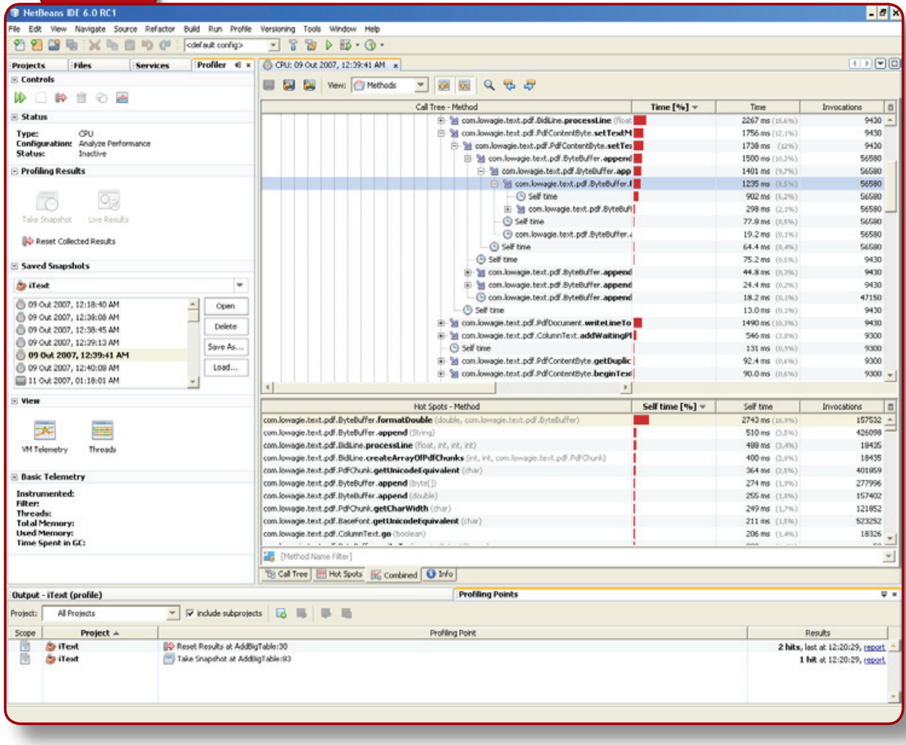
(◉) Entire application

Filter: Profile only project classes

[✓] Use defined Profiling Points

Figure 3. Starting the profiler

4



```
Status
Type:           CPU
Configuration:  Analyze Performance
Status:         Inactive

Saved Snapshots
09 Out 2007, 12:18:40 AM
09 Out 2007, 12:38:08 AM
09 Out 2007, 12:38:45 AM
09 Out 2007, 12:39:13 AM
09 Out 2007, 12:39:41 AM
09 Out 2007, 12:40:08 AM
11 Out 2007, 01:18:01 AM

Basic Telemetry
Instrumented:
Filter:
Threads:
Total Memory:
Used Memory:
Time Spent in GC:
```

CPU: 09 Out 2007, 12:39:41 AM   View: Methods

```
Call Tree - Method                                   Time [%]   Time                 Invocations
com.lowagie.text.pdf.BidiLine.processLine (float              2267 ms (15,6%)      9430
com.lowagie.text.pdf.PdfContentByte.setTextM                  1756 ms (12,1%)      9430
  com.lowagie.text.pdf.PdfContentByte.setTe                   1738 ms (12%)        9430
    com.lowagie.text.pdf.ByteBuffer.append                    1500 ms (10,3%)      56580
      com.lowagie.text.pdf.ByteBuffer.app                     1401 ms (9,7%)       56580
        com.lowagie.text.pdf.ByteBuffer.f                     1235 ms (8,5%)       56580
          Self time                                            902 ms (6,2%)       56580
          com.lowagie.text.pdf.ByteBuf                         298 ms (2,1%)       56580
        Self time                                              77.8 ms (0,5%)      56580
        com.lowagie.text.pdf.ByteBuffer.a                      19.2 ms (0,1%)      56580
      Self time                                                64.4 ms (0,4%)      56580
    Self time                                                  75.2 ms (0,5%)      9430
    com.lowagie.text.pdf.ByteBuffer.append                     44.8 ms (0,3%)      9430
    com.lowagie.text.pdf.ByteBuffer.append                     24.4 ms (0,2%)      9430
    com.lowagie.text.pdf.ByteBuffer.append                     18.2 ms (0,1%)      47150
  Self time                                                    13.0 ms (0,1%)      9430
com.lowagie.text.pdf.PdfDocument.writeLineTo                  1490 ms (10,3%)      9430
com.lowagie.text.pdf.ColumnText.addWaitingPl                   546 ms (3,8%)       9300
Self time                                                      131 ms (0,9%)       9300
com.lowagie.text.pdf.PdfContentByte.getDuplic                  92.4 ms (0,6%)      9300
com.lowagie.text.pdf.PdfContentByte.beginText                  90.0 ms (0,6%)      9300

Hot Spots - Method                                                                           Self time [%]   Self time          Invocations
com.lowagie.text.pdf.ByteBuffer.formatDouble (double, com.lowagie.text.pdf.ByteBuffer)                      2743 ms (18,9%)    157532
com.lowagie.text.pdf.ByteBuffer.append (String)                                                               510 ms (3,5%)     426098
com.lowagie.text.pdf.BidiLine.processLine (float, int, int, int)                                              488 ms (3,4%)     18435
com.lowagie.text.pdf.BidiLine.createArrayOfPdfChunks (int, int, com.lowagie.text.pdf.PdfChunk)                400 ms (2,8%)     18435
com.lowagie.text.pdf.PdfChunk.getUnicodeEquivalent (char)                                                     364 ms (2,5%)     401859
com.lowagie.text.pdf.ByteBuffer.append (byte[])                                                               274 ms (1,9%)     277996
com.lowagie.text.pdf.ByteBuffer.append (double)                                                               255 ms (1,8%)     157402
com.lowagie.text.pdf.PdfChunk.getCharWidth (char)                                                             249 ms (1,7%)     121852
com.lowagie.text.pdf.BaseFont.getUnicodeEquivalent (char)                                                     211 ms (1,5%)     523252
com.lowagie.text.pdf.ColumnText.go (boolean)                                                                  206 ms (1,4%)     18326
```

```
Project: All Projects   [✓] include subprojects

Scope  Project   Profiling Point                        Results
       iText     Reset Results at AddBigTable:30        2 hits, last at 12:20:29, report
       iText     Take Snapshot at AddBigTable:83        1 hit at 12:20:29, report
```

Figure 4. The NetBeans Profiler, focusing on method execution time

double value into a **String**, with up to six digits of decimal precision. But the code is huge. If a static flag, **HIGH_PRECISION**, is set to **true**, the method just relies on Java SE's **DecimalFormat** class. But if **HIGH_PRECISION==false** (the default), there follows a slab of code that's guaranteed to violate every size and complexity limit enforced by code validation tools.

Now, the obvious – and ironic – fact is that we've landed in code that was obviously optimized before. This is after all a mature project. What happens, one may ask, if **HIGH_PRECISION** is set to **true**? Not a pretty outcome. With HotSpot Server 6.0u3, the best individual running time goes from 562ms to 1,610ms! As in the original code, **formatDouble()** takes 20% of the total time, that means 112ms out of the total 562ms. With **HIGH_PRECISION==true**, as no other code is affected by this flag (I checked this), **formatDouble()** is consuming 72% of the total time. There's a 1,030% slowdown for a single method, and a 186% slowdown for the whole test.

The usual suspect

It's not surprising to find that **DecimalFormat** is a bottleneck. It's a well-known fact that **java.text**'s formatters are broken performance-wise. **DecimalFormat** objects are very expensive to create since their constructors compile the format strings into an optimized representation that makes each parsing/formatting operation very fast.

This is only good, though, if you can reuse each formatter for a large number of operations; but these objects are *not* thread-safe, so reuse opportunities are limited. The result is that any multithreaded application – even those with container-managed threads like Java EE apps – are forced to continuously recreate formatters, what can be very expensive.

The best is enemy of the good

Voltaire was hardly thinking about computer programming when he wrote this famous quote, but it serves us well. In the context of code optimization, almost any code can be enhanced a little further – but once any sufficiently complex code is "good enough", the additional optimization effort increases exponentially as you slowly approach the ideal of perfect, fastest-possible code.

The developer who noticed that **SimpleDateFormat** was chewing an insane amount of cycles may have jumped too early to the "perfect" solution: an extremely optimized, customized code that performs the required formatting as efficiently as physically possible in Java[3]. But is this really necessary? Aren't there alternative solutions which wouldn't lead to a method that's so massive and hard to maintain?

Analyzing iText's **ByteBuffer** class[4], you can see that it's not

Listing 3. The suspected ByteBuffer.formatDouble() (edited for space)

```
public static String formatDouble (double d, ByteBuffer buf) {
  if (HIGH_PRECISION) {
    // "Straight" formatting code
    DecimalFormat dn = new DecimalFormat("0.######", dfs);
    return dn.format(d);
  }
  // else... 200 lines(!) with custom formatting code.
}
```

Listing 4. Optimized ByteBuffer.formatDouble()

```
public class ByteBuffer extends OutputStream {
  ...
  private static final DecimalFormatSymbols dfs =
      new DecimalFormatSymbols(Locale.US);
  private DecimalFormat dnCached =
      new DecimalFormat("0.######", dfs);
  ...
  public static String formatDouble(double d, ByteBuffer buf){
    if (HIGH_PRECISION) {
      if (buf == null) {
        DecimalFormat dn = new DecimalFormat("0.######", dfs);
        return dn.format(d);
      }
      else {
        buf.append(buf.dnCached.format(d));
        return null;
      }
    }
  ...
}
```

[3] I don't know the history of iText's development, so this is really just a hypothesis that serves my reasoning here.

[4] From iText's **com.lowagie.text.pdf** package; no relation to **java.nio.ByteBuffer**.

thread-safe, because of several mutable fields and no synchronization. This is the common design of most buffer-esque objects: they are not often shared, and if they are, they're better synchronized in upper layers. But this means that the optimization shown in **Listing 4** is valid. Here I created a shared **DecimalFormat** object, limiting the sharing to multiple invocations on the same **ByteBuffer** instance. As the buffers are never shared, the thread-safety limitation of **DecimalFormat** is meaningless. I performed the tests again with **HIGH_PRECISION==true**. The result was 1,047ms; much better than the original result with this option set, but still a significant slowdown (86% worse) over the score for **HIGH_PRECISION==false**.

This **formatDouble()** method is tricky: though static, it can be invoked with a **ByteBuffer** argument – an "optional **this**". If a buffer is received, the formatted value is appended to the buffer; otherwise (i.e. **null** is passed), the formatted value is returned. So a good hypothesis is that I didn't obtain all the speedup I wished for, because there are many invocations with **buf==null**, and I couldn't optimize this case to reuse the formatter.

Back to the Profiler

Profilers are not only good for finding bottlenecks and performance bugs. They are also great for validating and refining your findings in an iterative and interactive process that should lead to the desired code enhancement.

In the same *Call Tree* page shown in **Figure 4**, right click the **formatDouble()** method and choose *Show Back Traces*. Now you have a new tab labeled "*Back traces for: formatDouble*", showing all code paths that end in that method (see **Figure 5**). You'll see two branches. The top one shows traces where the immediate caller of **formatDouble()** is the **append(double)** instance method, which passes **this** to its parameter **buf**. In the bottom branch, the caller is a static method **formatDouble(double)**, which passes **null** for this parameter.

We could imagine that the second branch is guilty for the remaining slowness; but a quick look at the numbers proves this not to be true. Even though each invocation in the "slow path" is very expensive, there are very few such invocations – a total of 130, compared to 157,532 invocations in the "fast path". The "*Time [%]*" column in the same tab confirms that virtually all execution time goes to the fast-path branch.

This means that the remaining slowness is not caused by an excessive number of **DecimalFormat** instantiations. It comes from the execution of this object's **format()** method, which is still slower than the highly customized code that **ByteBuffer.formatDouble()** has for the **HIGH_PRECISION==false** case.

Correct? Perhaps. There is one potential flaw in this conclusion: we don't know how many **ByteBuffer** objects are being created. Consequently we don't know how many times our optimized instance field (**DecimalFormat dnCached**) is being created. It's time for a different profiling strategy.

Profiling memory allocation

Start the Profiler again, now with *Profile>Profile Main Project, Memory*. Accept the defaults for all options and run a new profiling session. Now we have more interesting results: each test iteration allocates 9,440 instances of **ByteBuffer** and 9,571 instances of **DecimalFormat** (this is after optimization – remember that the shared formatting object is only used for one of two possible code paths into **formatDouble()**). The original unoptimized code would allocate 166,973 instances of **DecimalFormat** per iteration.

I managed to cut 95% of these allocations (as well as the expensive constructions involved), so my instincts say there's not much to be gained by cutting the remaining 5%. The remaining cost of the simpler formatting code should come from the *execution* of **DecimalFormat.format()**, not from the construction of **DecimalFormat** objects.

That's game over for our analysis of **formatDouble()**. The method is already very close to an optimum implementation if **DecimalFormat** is to be used. See the

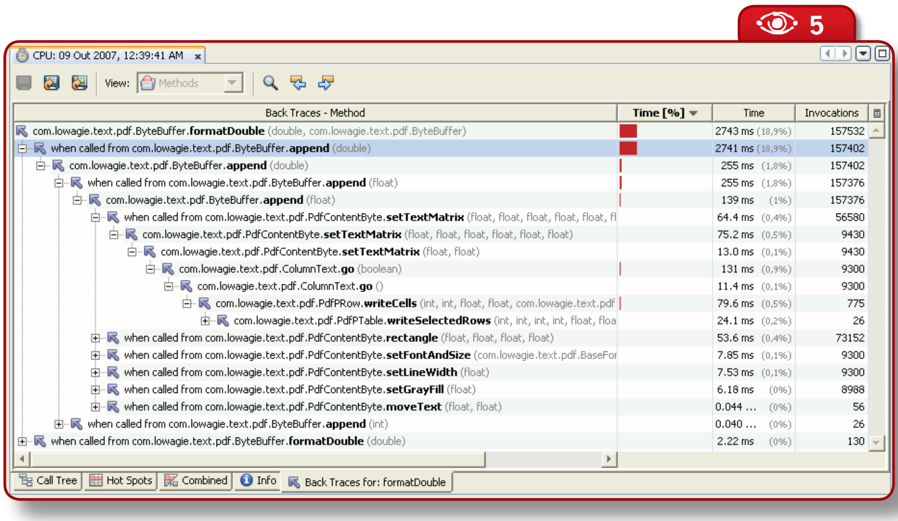
[5] Remember that Java 5's enhanced-for does not avoid iterators; it only hides them from you: **for (T item: items)** will walk the **items** collection with an iterator. Incidentally, iterators are great candidates to be "lightweight objects". This new language feature may appear in future JVMs; see John Rose's Multi-language VM proposal at *mail.openjdk.java.net/pipermail/announce/2007-October/000016.html*. Another solution (specific to iterators and other objects that are typically used in local scope) is automatic stack allocation (driven by escape analysis), an optimization the HotSpot team is researching but which has not been implemented in production JVMs from Sun up to now.

box “Next step for formatDouble()?” for additional findings that are a final part of the optimization of that method – but follow me here in the investigation of the NetBeans Profiler.

The *Memory Results* view in **Figure 6** shows the top allocations, which may be good enough for solving some performance diagnostics. Sometimes you'll also need to consider the containment relationships between several classes: for example, most **char[]** objects are private fields of **String** or **StringBuffer**/**StringBuilder** objects, and all **HashMap$Entry** objects are used inside **HashMap** instances. So you can easily spot some common behaviors of iText; for example it creates a large number of **String**s – which is not surprising since it's a document-processing library. What seems less natural is that iText also allocates a good number of **AbstractList$Itr** objects. This doesn't look like an unavoidable cost of the task being performed.

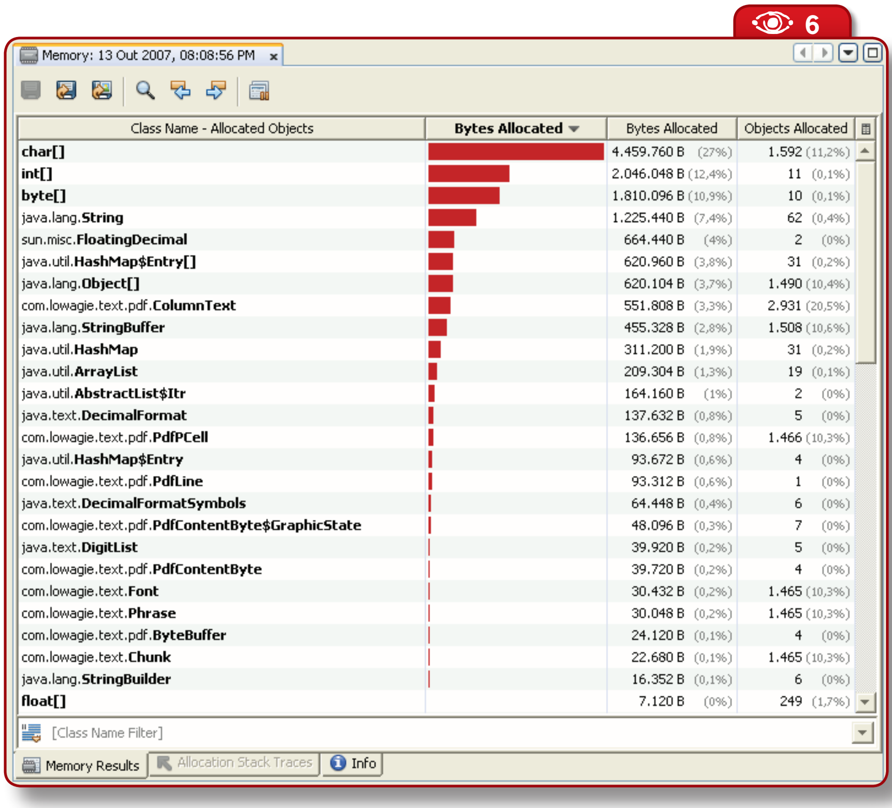
We've found a place that deserves further inspection. We'll need require additional profiling data: run a new memory profiling session, this time activating the option *Record stack trace for allocation*. Now, for each class listed in the *Memory Results* tab, the NetBeans Profiler enables a context menu item: *Show Allocation Stack Traces*.

In **Figure 7** you can see all code locations where iterators are allocated. Java's iterators often cause performance problems, because they are heap-allocated objects. They create additional costs as well: their “fail-fast” behavior makes their implementation



Back Traces - Method	Time [%]	Time	Invocations
com.lowagie.text.pdf.ByteBuffer.**formatDouble** (double, com.lowagie.text.pdf.ByteBuffer)		2743 ms (18,9%)	157532
when called from com.lowagie.text.pdf.ByteBuffer.**append** (double)		2741 ms (18,9%)	157402
com.lowagie.text.pdf.ByteBuffer.**append** (double)		255 ms (1,8%)	157402
when called from com.lowagie.text.pdf.ByteBuffer.**append** (float)		255 ms (1,8%)	157376
com.lowagie.text.pdf.ByteBuffer.**append** (float)		139 ms (1%)	157376
when called from com.lowagie.text.pdf.PdfContentByte.**setTextMatrix** (float, float, float, float, float, fl		64.4 ms (0,4%)	56580
com.lowagie.text.pdf.PdfContentByte.**setTextMatrix** (float, float, float, float, float, float)		75.2 ms (0,5%)	9430
com.lowagie.text.pdf.PdfContentByte.**setTextMatrix** (float, float)		13.0 ms (0,1%)	9430
com.lowagie.text.pdf.ColumnText.**go** (boolean)		131 ms (0,9%)	9300
com.lowagie.text.pdf.ColumnText.**go** ()		11.4 ms (0,1%)	9300
com.lowagie.text.pdf.PdfPRow.**writeCells** (int, int, float, float, com.lowagie.text.pdf		79.6 ms (0,5%)	775
com.lowagie.text.pdf.PdfPTable.**writeSelectedRows** (int, int, int, int, float, floa		24.1 ms (0,2%)	26
when called from com.lowagie.text.pdf.PdfContentByte.**rectangle** (float, float, float, float)		53.6 ms (0,4%)	73152
when called from com.lowagie.text.pdf.PdfContentByte.**setFontAndSize** (com.lowagie.text.pdf.BaseFor		7.85 ms (0,1%)	9300
when called from com.lowagie.text.pdf.PdfContentByte.**setLineWidth** (float)		7.53 ms (0,1%)	9300
when called from com.lowagie.text.pdf.PdfContentByte.**setGrayFill** (float)		6.18 ms (0%)	8988
when called from com.lowagie.text.pdf.PdfContentByte.**moveText** (float, float)		0.044 ... (0%)	56
when called from com.lowagie.text.pdf.ByteBuffer.**append** (int)		0.040 ... (0%)	26
when called from com.lowagie.text.pdf.ByteBuffer.**formatDouble** (double)		2.22 ms (0%)	130

Figure 5. The Back Traces tab



Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
char[]		4.459.760 B (27%)	1.592 (11,2%)
int[]		2.046.048 B (12,4%)	11 (0,1%)
byte[]		1.810.096 B (10,9%)	10 (0,1%)
java.lang.**String**		1.225.440 B (7,4%)	62 (0,4%)
sun.misc.**FloatingDecimal**		664.440 B (4%)	2 (0%)
java.util.**HashMap$Entry[]**		620.960 B (3,8%)	31 (0,2%)
java.lang.**Object[]**		620.104 B (3,7%)	1.490 (10,4%)
com.lowagie.text.pdf.**ColumnText**		551.808 B (3,3%)	2.931 (20,5%)
java.lang.**StringBuffer**		455.328 B (2,8%)	1.508 (10,6%)
java.util.**HashMap**		311.200 B (1,9%)	31 (0,2%)
java.util.**ArrayList**		209.304 B (1,3%)	19 (0,1%)
java.util.**AbstractList$Itr**		164.160 B (1%)	2 (0%)
java.text.**DecimalFormat**		137.632 B (0,8%)	5 (0%)
com.lowagie.text.pdf.**PdfPCell**		136.656 B (0,8%)	1.466 (10,3%)
java.util.**HashMap$Entry**		93.672 B (0,6%)	4 (0%)
com.lowagie.text.pdf.**PdfLine**		93.312 B (0,6%)	1 (0%)
java.text.**DecimalFormatSymbols**		64.448 B (0,4%)	6 (0%)
com.lowagie.text.pdf.**PdfContentByte$GraphicState**		48.096 B (0,3%)	7 (0%)
java.text.**DigitList**		39.920 B (0,2%)	5 (0%)
com.lowagie.text.pdf.**PdfContentByte**		39.720 B (0,2%)	4 (0%)
com.lowagie.text.**Font**		30.432 B (0,2%)	1.465 (10,3%)
com.lowagie.text.**Phrase**		30.048 B (0,2%)	1.465 (10,3%)
com.lowagie.text.pdf.**ByteBuffer**		24.120 B (0,1%)	4 (0%)
com.lowagie.text.**Chunk**		22.680 B (0,1%)	1.465 (10,3%)
java.lang.**StringBuilder**		16.352 B (0,1%)	6 (0%)
float[]		7.120 B (0%)	249 (1,7%)

Figure 6. Top allocated objects by heap usage

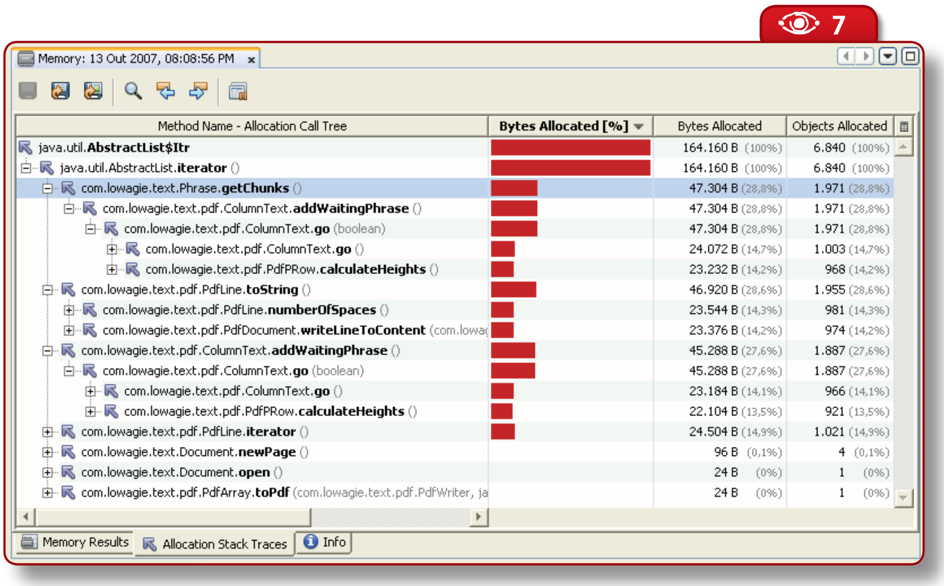


Figure 7. Allocation stack traces for list iterators

Memory: 13 Out 2007, 08:08:56 PM

Method Name - Allocation Call Tree	Bytes Allocated [%]	Bytes Allocated	Objects Allocated
java.util.**AbstractList$Itr**	🟥	164.160 B (100%)	6.840 (100%)
java.util.AbstractList.**iterator** ()	🟥	164.160 B (100%)	6.840 (100%)
com.lowagie.text.Phrase.**getChunks** ()	🟥	47.304 B (28,8%)	1.971 (28,8%)
com.lowagie.text.pdf.ColumnText.**addWaitingPhrase** ()	🟥	47.304 B (28,8%)	1.971 (28,8%)
com.lowagie.text.pdf.ColumnText.**go** (boolean)	🟥	47.304 B (28,8%)	1.971 (28,8%)
com.lowagie.text.pdf.ColumnText.**go** ()	🟥	24.072 B (14,7%)	1.003 (14,7%)
com.lowagie.text.pdf.PdfPRow.**calculateHeights** ()	🟥	23.232 B (14,2%)	968 (14,2%)
com.lowagie.text.pdf.PdfLine.**toString** ()	🟥	46.920 B (28,6%)	1.955 (28,6%)
com.lowagie.text.pdf.PdfLine.**numberOfSpaces** ()	🟥	23.544 B (14,3%)	981 (14,3%)
com.lowagie.text.pdf.PdfDocument.**writeLineToContent** (com.lowa	🟥	23.376 B (14,2%)	974 (14,2%)
com.lowagie.text.pdf.ColumnText.**addWaitingPhrase** ()	🟥	45.288 B (27,6%)	1.887 (27,6%)
com.lowagie.text.pdf.ColumnText.**go** (boolean)	🟥	45.288 B (27,6%)	1.887 (27,6%)
com.lowagie.text.pdf.ColumnText.**go** ()	🟥	23.184 B (14,1%)	966 (14,1%)
com.lowagie.text.pdf.PdfPRow.**calculateHeights** ()	🟥	22.104 B (13,5%)	921 (13,5%)
com.lowagie.text.pdf.PdfLine.**iterator** ()	🟥	24.504 B (14,9%)	1.021 (14,9%)
com.lowagie.text.Document.**newPage** ()		96 B (0,1%)	4 (0,1%)
com.lowagie.text.Document.**open** ()		24 B (0%)	1 (0%)
com.lowagie.text.pdf.PdfArray.**toPdf** (com.lowagie.text.pdf.PdfWriter, ja		24 B (0%)	1 (0%)

more complex, requiring additional indirections and polymorphic calls inside loops. That's why, in performance critical code, I avoid iterators like the plague[5].

Most iterators identified in **Figure 7** are being created gratuitously, in methods like **Phrase.getChunks()** and **PdfLine.toString()**. The static type of iterated collections is always **ArrayList**, and iterators are used just for walking the collections (there are no other operations, like **remove()**). This shows that iText is optimized to avoid using collection interfaces (like **List**), when such flexibility is not necessary. In this case, why use iterators? Using loops with **get()** and **size()** would be faster.

A good excuse for iterators is dealing with collections whose static type is an interface, because you either don't have access to more specialized access methods, or aren't sure of their performance traits. In particular, iterating an arbitrary **List** with indexing is a potential disaster, because if the list happens to be a **LinkedList**, the random access methods are available but they'll crawl, of course.

Both of these top iterator-allocating methods could benefit from indexed access and other optimizations. See **Listing 5**. The **toString()** method has two small problems. First, it uses **StringBuffer** (instead of the much better **StringBuilder** API of Java SE 5.0+); this is probably a concession to compatibility with older runtimes. Second, **toString()** does not preallocate the size of the buffer, which is difficult to estimate, because each **PdfChunk.toString()** may return a string with a different size. But in my experience, even a very raw and conservative estimation – say **line.size()*16**, where 16 is a (somewhat arbitrary) small size per chunk – is much better than no estimation at all (which often causes excessive reallocation).

There's a similar problem in the **getChunks()** method: the new **ArrayList** that accumulates elements taken from the current object (**Phrase** itself is a list) lacks preallocation. This case could also benefit from a conservative estimation, e.g. **new ArrayList(size())**.

So there you go – no less than five optimization opportunities in just two methods totalling 14 lines of code. Should we go ahead and execute all five? This requires, as usual, some additional thought. Of course I'd need to have more experience with the iText codebase to determine

Listing 5. Methods deserving small optimizations

```
// class PdfLine:
public String toString() {
  StringBuffer tmp = new StringBuffer();
  for (Iterator i = line.iterator(); i.hasNext(); ) {
    tmp.append(((PdfChunk) i.next()).toString());
  }
  return tmp.toString();
}

// class Phrase:
public ArrayList getChunks() {
  ArrayList tmp = new ArrayList();
  for (Iterator i = iterator(); i.hasNext(); ){
    tmp.addAll(((Element) i.next()).getChunks());
  }
  return tmp;
}
```

Next step for formatDouble()?

In the investigation of iText summed up in the body of the article, I was able to optimize **ByteBuffer.formatDouble()**'s "straight" formatting code significantly (using **SimpleDateFormat**). But not enough to compete with the performance of the existing optimized formatter.

Profiling and optimization are hard, but even more difficult is making tradeoffs. Specifically, is the optimized code good enough to be preferable – if not by default, then at least in some circumstances – over the existing custom formatter? There's no easy answer.

The original straight code was clearly unacceptable. You'll notice, if you read iText's full *ByteBuffer.java*, that the **HIGH_PRECISION** flag has its value hardwired. The shipping iText binary is compiled with **HIGH_PRECISION==false**, and this option can only be changed by editing the source code and recompiling the library.

Now, the optimized straight code makes my benchmark only 86% slower (instead of 186%) than the custom format code. This is still significantly slower, so certainly iText users would prefer to keep using the existing optimized formatter. Or not? Speed is not everything, and the name of the **HIGH_PRECISION** flag implies of course that when it's set to **false** some loss of precision is expected. Indeed, the optimized formatter performs some approximations like this one (edited):

```
if (Math.abs(d) < 0.000015) {
  return "0";
}
```

This truncates many close-to-zero numbers, killing their sixth decimal position (plus roughly one bit of the fifth). There are additional approximations in the code.

An experiment talks louder than any hypothesis, so I executed the program twice, first with **HIGH_PRECISION** set to **false** and then setting the flag to **true** – with both runs writing the PDF content to a file as the original code did. The default low-precision setting produced a document with 119,262 bytes, but the high-precision document was significantly bigger: 135,007 bytes. Then I instrumented the program to format each value with both algorithms and dump the results. To my surprise, I saw that the optimized algorithm used only two decimal digits of precision for all numbers! Here's a typical output for this instrumented run:

```
hi: 91.333344, lo: 91.33
hi: 18, lo: 18
hi: 1, lo: 1
hi: 333.818176, lo: 333.82
```

I discovered that my comparison is not fair. I'm comparing an algorithm that formats with six decimal digits of precision against one that goes only to two decimal places.

I fixed this by simply changing the format string to "0.##". Touché: the PDF was created with 119,262 bytes, identical to the optimized formatter (which is also a good validation of our new code if we consider **DecimalFormat** as the "canonical" formatter). As a result, the execution time went down to 984ms. This is still 75% worse than the optimized formatter, so it doesn't change matters a lot…

Unless the full precision is useful. If PDF files contain a large number of floating-point numbers, I'd expect additional precision to have an impact on the quality of the documents. I tried to measure this, but without success; the "high-precision" PDF looked identical to the "low-precision" one to my bare eyes (on 1280x1024 resolution and using Adobe Reader 8.1).

Perhaps the problem is that the test document is too simple for us to detect the difference – it's just a huge table with heading and borders, and loads of dummy alphanumerical data filling the cells. But a program that produces PDFs with complex, high-precision graphics – say, a CAD tool – may result in a perceivable advantage for the high-precision flag, especially when printing the PDF on a high-DPI printer or plotter. I will have to leave this conclusion to iText's developers and advanced users.

If we consider that the high-precision output could be useful in some scenarios, the enhanced formatter might be a good option even with a significant speed hit. In this case I'd leave **HIGH_PRECISON==false** as default, but provide some means of changing it. (It's also possible that I picked a sample that depends on **formatDouble()**'s performance much more than usual.)

if each of these optimizations is worth its cost (e.g., the cost of having less maintainable/readable code, *if* you think that iterators are better than explicit indexed access).

In any case, this is a good opportunity to remember that efficient programs are not created only with a handful of super optimizations with order-of-magnitude gains (also known as "having fun"). They are also created with hundreds of tiny optimizations which in aggregate can make a very significant difference.

Walking the heap

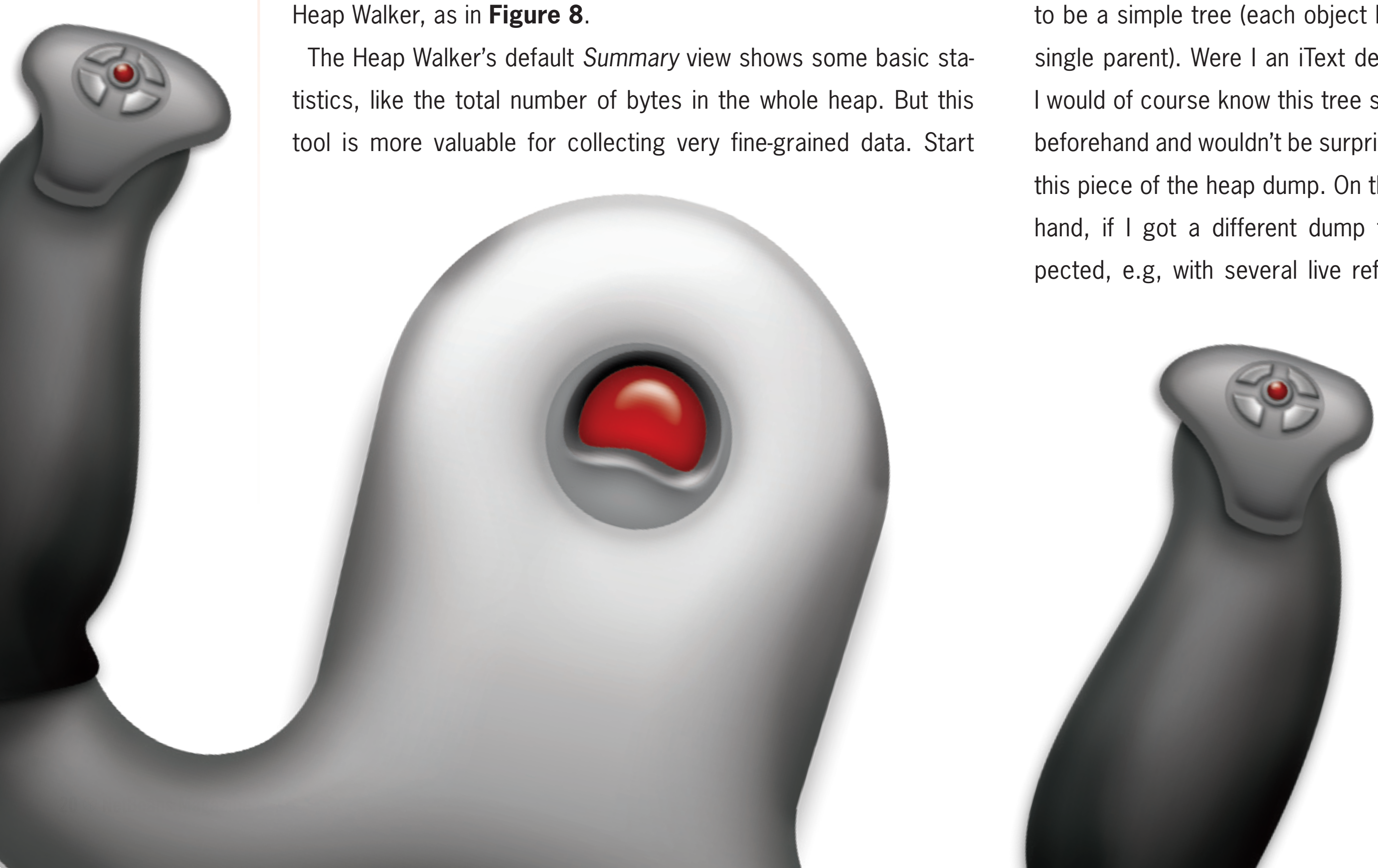
The **Heap Walker** is another major new feature of the Profiler in NetBeans 6.0. Performance problems can often be diagnosed on a relatively high-level, with coarse summary information like "too many instances of **X** are being allocated". But sometimes this isn't enough; you need finer-grained data like "too many instances of **X** are allocated as fields of **Y**". This leads to diagnostics like "perhaps I should lazy-initialize these fields since they're rarely used".

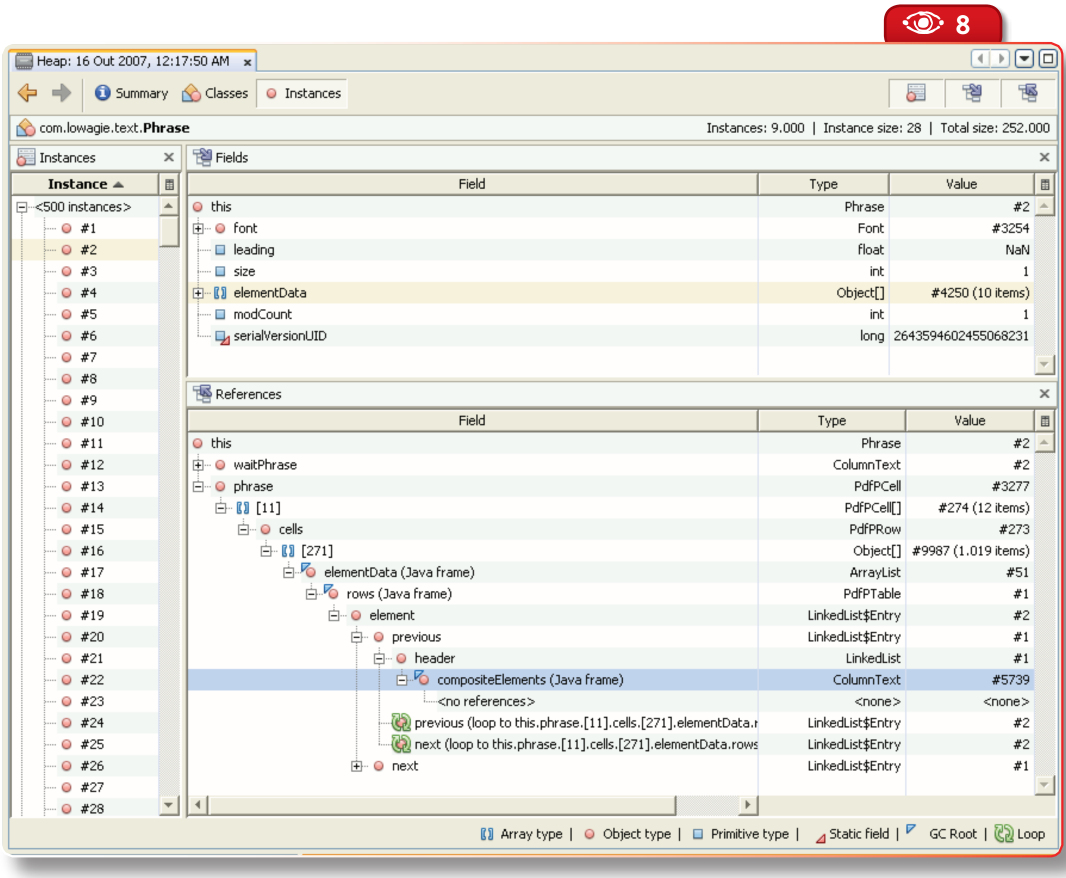
To help handling these situations is the role of the Heap Walker. In order to test this feature, run a new profiling session (any profiling mode will do) and in the middle of the program's execution run *Profile>Take Heap Dump*. Pick *Profiled project* as the destination, stop the profiling session and open the heap dump (it's available in the *Saved snapshots* window). Heap dumps will be opened by the Heap Walker, as in **Figure 8**.

The Heap Walker's default *Summary* view shows some basic statistics, like the total number of bytes in the whole heap. But this tool is more valuable for collecting very fine-grained data. Start selecting the *Classes* view. This shows a tabulation of instance counts and heap sizes per class, similar to the *Memory Results* tab of the memory profiling snapshots. The difference is that now you can select a class and right click *Show in Instances view*.

This brings a three-pane view. The *Instance* pane lists every single instance of the class (instances may be grouped in blocks of 500). Select any individual instance, and you see its fields in the *Fields* pane and the objects that point to it in the *References* pane. This information about references is usually the most interesting.

For example, when inspecting a random **Phrase** object from iText, you'll be able to see the whole tree of objects forming the PDF document's object model. You can verify that a certain **Phrase** object is contained by a **PdfPCell[]** array, which is in a **PdfPRow** object, and so forth. The structure below the **PdfPCell[]** array seems to be a simple tree (each object having a single parent). Were I an iText developer, I would of course know this tree structure beforehand and wouldn't be surprised with this piece of the heap dump. On the other hand, if I got a different dump than expected, e.g, with several live references





com.lowagie.text.**Phrase** Instances: 9.000 | Instance size: 28 | Total size: 252.000

Instance ▲
<500 instances>
#1
#2
#3
#4
#5
#6
#7
#8
#9
#10
#11
#12
#13
#14
#15
#16
#17
#18
#19
#20
#21
#22
#23
#24
#25
#26
#27
#28

Fields

```
Field                                                    Type        Value
this                                                     Phrase      #2
+ font                                                   Font        #3254
  leading                                                float       NaN
  size                                                   int         1
+ elementData                                            Object[]    #4250 (10 items)
  modCount                                               int         1
  serialVersionUID                                       long        2643594602455068231
```

References

```
Field                                                    Type              Value
this                                                     Phrase            #2
+ waitPhrase                                             ColumnText        #2
- phrase                                                 PdfPCell          #3277
  - [11]                                                 PdfPCell[]        #274 (12 items)
    - cells                                              PdfPRow           #273
      - [271]                                            Object[]          #9987 (1.019 items)
        - elementData (Java frame)                       ArrayList         #51
          - rows (Java frame)                            PdfPTable         #1
            - element                                    LinkedList$Entry  #2
              - previous                                 LinkedList$Entry  #1
                - header                                 LinkedList        #1
                  - compositeElements (Java frame)       ColumnText        #5739
                      <no references>                    <none>            <none>
                  previous (loop to this.phrase.[11].cells.[271].elementData.r  LinkedList$Entry  #2
                  next (loop to this.phrase.[11].cells.[271].elementData.rows   LinkedList$Entry  #2
              + next                                     LinkedList$Entry  #1
```

[] Array type | ● Object type | ■ Primitive type | ◢ Static field | ◤ GC Root | Loop

Figure 8. The Heap Walker

to the same **PdfPRow** object, this could be evidence of some problem, such as a memory leak or bugs in the document construction algorithms.

Indeed, heap walking tools are often as useful for debugging as they are for profiling, if not even more handy.

Conclusions

The focus of this article was to illustrate profiling techniques on a real-world example and concentrate on the major new features in the NetBeans 6.0 Profiler: Profiling Points and the Heap Walker. (Incidentally, there are many other improvements we didn't cover, such as comparison of memory snapshots, drill-down graphs, integration with JMeter, and support for JDK 6's dynamic attachment facility.)

As you could see from our sample profiling sessions, code profiling and optimization can be a laborious, trial-and-error process. You may need to write new code just to support this work, even if you have access to the best tools; then you'll spend most of your time thinking about the consequences of each code tweak. In my experience, the actual code optimization is often the easiest part. It's often obvious, once you *fully* understand the problem.

Writing high performance code is difficult, but it is much easier with good tools. And the NetBeans Profiler is one of the best tools anywhere for that job: it automates repetitive tasks like taking snapshots at the right moment, runs efficiently so as to not break your creative mood with long pauses, and makes a lot of data easily accessible through clear visualizations.

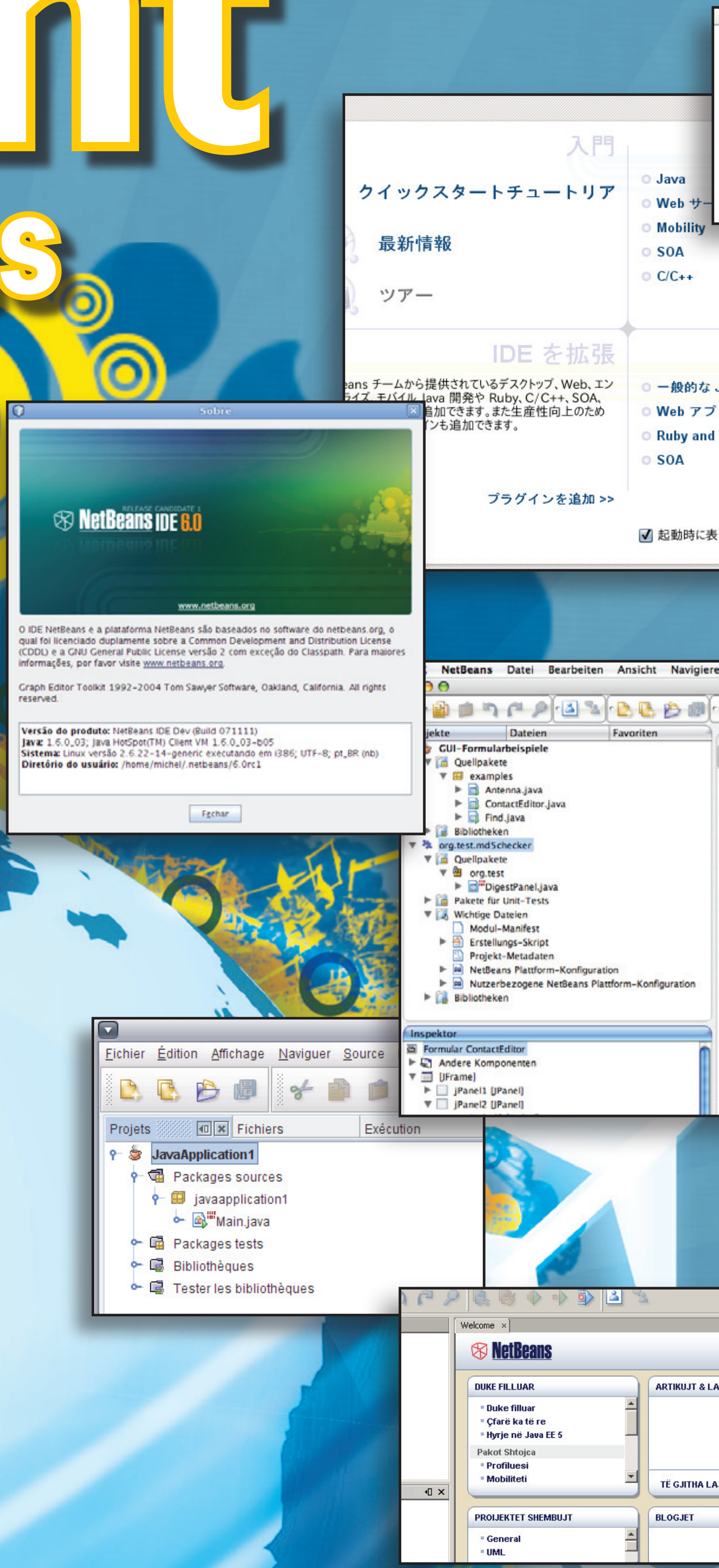
If you go through the steps in this article following it as a tutorial, you'll notice that the time spent in the Profiler (e.g., running a particular command) is minimal. That's the mark of a great tool: not getting in your way – because you're paid to build engines, not to use screwdrivers.

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

Fluent in NetBeans

Spreading the IDE to Many Worlds

Janice Campbell



eans チームから提供されているデスクトップ、Web、エン
ライズ、モバイル Java 開発や Ruby、C/C++、SOA、
追加できます。また生産性向上のため
インも追加できます。

O IDE NetBeans e a plataforma NetBeans são baseados no software do netbeans.org, o qual foi licenciado duplamente sobre a Common Development and Distribution License (CDDL) e a GNU General Public License versão 2 com exceção do Classpath. Para maiores informações, por favor visite www.netbeans.org.

Graph Editor Toolkit 1992-2004 Tom Sawyer Software, Oakland, California. All rights reserved.

```
Versão do produto: NetBeans IDE Dev (Build 071111)
Java: 1.6.0_03; Java HotSpot(TM) Client VM 1.6.0_03-b05
Sistema: Linux versão 2.6.22-14-generic executando em i386; UTF-8; pt_BR (nb)
Diretório do usuário: /home/michel/.netbeans/6.0rc1
```

How a global community is making NetBeans more accessible to non-English speaking developers – one language at a time

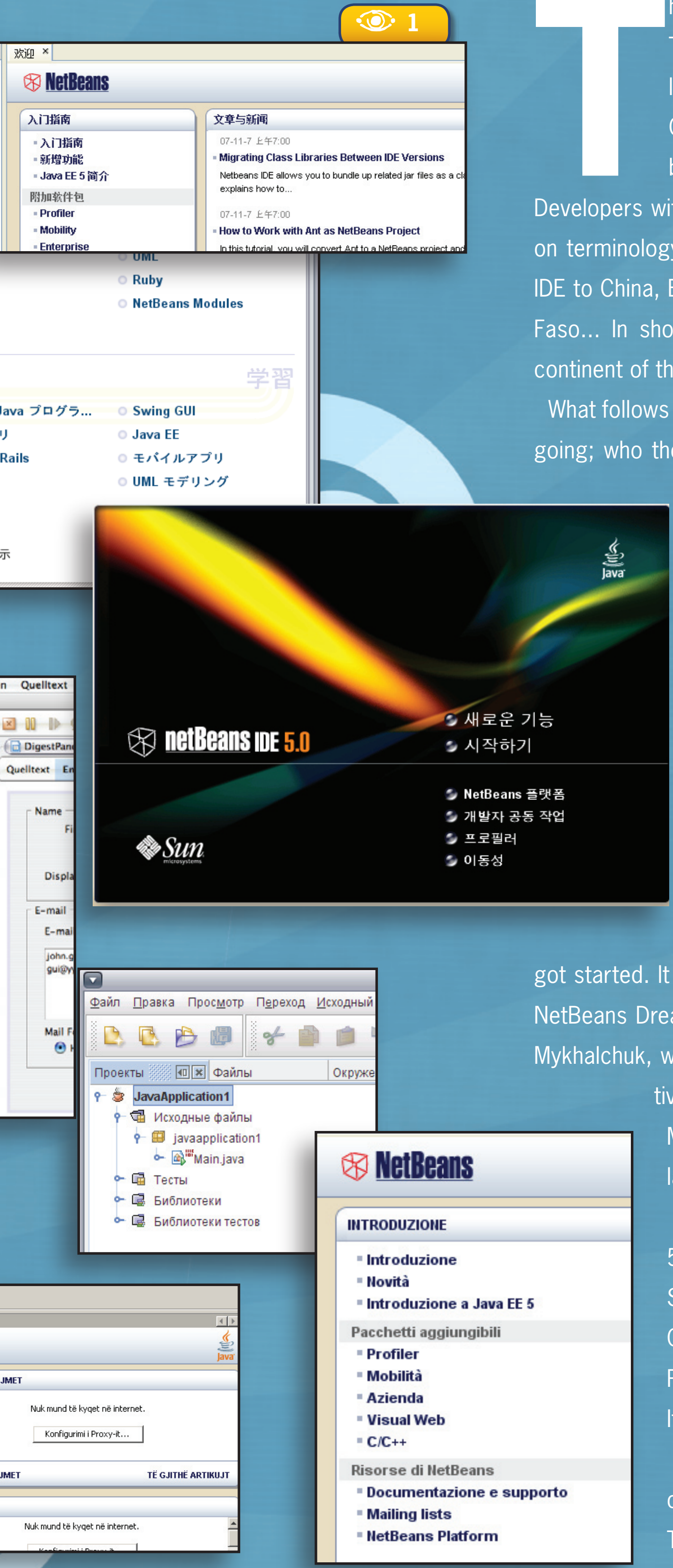


Figure 1. NetBeans 5.0, 5.5, 5.5.1 and 6.0 in Simplified Chinese, Portuguese, Italian, Korean, Albanian, Japanese, Russian, French and German

The greatest asset NetBeans has is its community. They are the end-users who contribute to its excellence by giving back to the product in so many ways. One community group is making a big difference by bringing NetBeans to the non-English speaking world. Developers with multilingual skills contribute translations, consult on terminology, and offer their technical expertise to deliver the IDE to China, Brazil, Russia, Indonesia, Belgium, Panama, Burkina Faso… In short, they reach out to developers on nearly every continent of the globe.

What follows is a history of how the project started and where it's going; who the pioneers were; some problems encountered and how they were solved; and how to participate or get started translating NetBeans into your language.

How did it all start?

A long way back, at the time of NetBeans version 3.4 or 3.5, a couple of motivated NetBeans aficionados realized that the tool had the potential to reach and benefit a much wider user group if it were translated into languages other than English. That's how the TranslatedFiles (*translatedfiles.netbeans.org*) community project got started. It was born out of the vision of Vincent Brabant (now NetBeans Dream Team member and Java Champion) and Maxym Mykhalchuk, who produced French and Russian versions, respectively. At version 4.0, Dutch-speaking Java Champion Manfred Riem joined the effort, adding another language to the mix.

Since that time, modules of the NetBeans IDE 5.0, 5.5, or 5.5.1 have been translated into Albanian, Simplified and Traditional Chinese, Azerbaijani, Czech, French, German, Spanish, Korean, Dutch, Russian, Brazilian Portuguese, Swedish, Indonesian, Italian, and Japanese.

Now, with the launch of NetBeans 6.0 translations, contributors from new languages such as Bulgarian, Turkish, and Polish have recently joined the project. **Figure 1** gives a hint of the various translated versions of NetBeans IDE releases.

netbeans.org/community/articles/zh-tw55-release.html

The Traditional Chinese team and NetBeans release 5.5

Who is carrying on the legacy?

The Software Globalization business unit at Sun joined the project around NetBeans 4.x in order to support the Japanese and Simplified Chinese versions. Translations were added for Help files, tutorials, and product and web documentation. With each new version of NetBeans, more and more community members with linguistic skills joined the project and grew the number of languages being worked on. Over time, translation contributions were being received for the various netbeans.org community pages as well as product documentation.

Thanks to the strong influence of SouJava, Brazil's largest Java User Group, a Brazilian Portuguese team expanded this year to be the largest in membership size. This led to a pioneering collaboration between Sun translators and the Brazil community contributors. A highly-motivated and dedicated team of in-country developers, translating during their spare time, succeeded in releasing a high-quality NetBeans 5.5 in Brazilian Portuguese on the same schedule as the official Sun multi-lingual version. **Figure 2** shows a core set of the team.

To get a feel for the people behind the scenes, let's take a closer look at some of the teams.

"Participating in this community is a way to help spread NetBeans among Spanish-speaking developers and students. It has also allowed me to meet and be a part of a translation team that includes members from other countries such as Colombia, Mexico, Spain, Argentina, Cuba, Guatemala, Peru, Chile, Venezuela, and of course, Panama."

Aristides Villarreal, NetBeans Dream Team member and PanamaJUG founder

- **German** – Whenever Ruth Kusterer isn't doing her day job as a member of the netbeans.org web team, she's recruiting new members to the German localization group and helping in the translation, review, and CVS commit activities, alongside Peter, Holger, Jake and others. The half dozen or so members hail from Germany, Austria or Switzerland, and have contributed translations for the IDE as well as the UML module.
- **Spanish –** The Spanish-language team goes back a couple of releases, even before Argentinean Diego Gil and Spaniard David Álvarez León created the Spanish mailing list. More recently, the PanamaJUG, under the leadership of Aristides Villarreal, has been instrumental in growing the Spanish-language community throughout Latin

2

Figure 2. Some Brazil team members in attendance at Sun Tech Days São Paulo

America and Spain. A Spanish-language version has been released for NetBeans 5.5 and 5.5.1; NetBeans 6.0 Spanish is in progress.

- **Polish –** The Polish team is one of the newest localization groups. Magda Niedzwiecka Goldyn recently joined Sun and started immediately making contacts with universities and Java User Groups in Poland. She created a NetBeans discussion list (*nbdiscuss_pl@netbeans.org*) and met a number of enthusiastic Polish subscribers, like Paul, Jacek, Leszek and others, who were already organizing to start a translation effort for NetBeans 6.0. **Figure 3** shows a translated NetBeans 6.0 dialog.

What bodes for the future? As government entities and public universities around the world migrate to open source software, we are sure to see an increased interest in new localized versions for NetBeans. With each new release, teams expand and the number of languages grows.

Why do they contribute?

If joining and contributing to a project requires a lot of extra time outside work, studies, and home life, why do people do it? Numerous reasons can be cited. Some people enjoy the creative and challenging process of developing software; while others are interested in building a reputation and expertise in related technologies. And of course, community affiliation compels people to seek association with like-minded individuals who enjoy sharing ideas and experiences, and building communities and friendships.

When asked about their reasons for participating in these community projects, some TranslatedFiles contributors said they were strongly committed to the free and open-source software ideology. Others believe that it is only fair to give back to a community from which one has derived benefits.

Most would agree that making the tool more accessible to developers – rendering it in their native languages – is a key rationale for joining the project. For example, translating, at a minimum, the NetBeans Platform allows developers to write modules and build rich-client applications on top of a localized version of the Platform. The application is then ready to be deployed in the native language.

"I like to participate in the project because it's an opportunity to get to know new people from Brazil and other countries. Exchanging experiences always improves our technical and personal skills. People feel proud to take part."

Jefferson Prestes, Brazilian contributor since version 5.0

Henrique Meira said it best in a motivational e-mail to his Brazilian team members, when he enumerated the benefits of contributing to NetBeans 6.0 translations: "We have Sun's management support and free access to worldwide events, technical assistance and a social community, and awards are given to collaborators – but most importantly we give back to NetBeans because of its excellence."

What problems do they face?

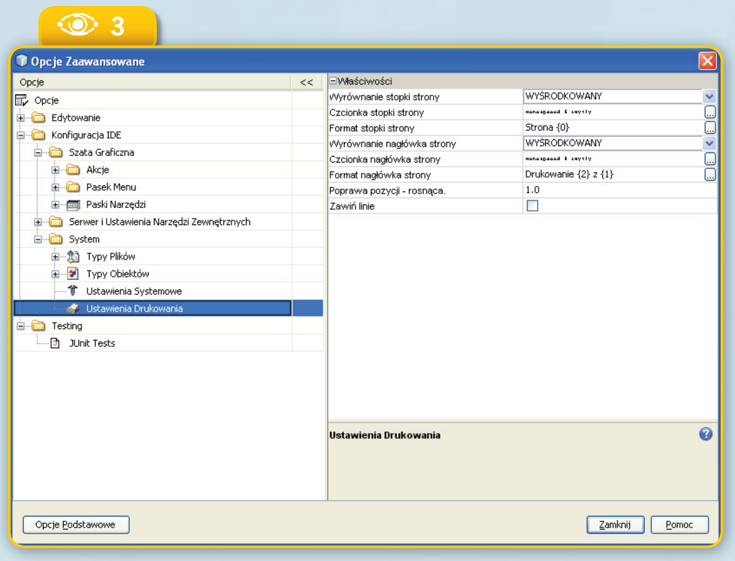
Time is the single most difficult constraint that all the contributors face. It is assumed that everyone participating is a full-time developer, employee, student, or maybe professor with jobs, families and a social life. That means most of the work has to be accomplished in between commitments. As is the case in many collaborative projects, the more members one can bring to the project, the more workload that can be spread over a larger pool of people, so that no single individual feels responsible for making

translatedfiles.netbeans.org

NetBeans localization/ translation community project

netbeans.org/community/articles/interviews/ifnu-bima-indonesia-nug.html

Article about the Indonesian NetBeans Users Group and the NetBeans localization efforts of NUG members Ibrahim F. Burhan and Thomas Edwin Santosa



Właściwości

Wyrównanie stopki strony	WYŚRODKOWANY
Czcionka stopki strony	[illegible]
Format stopki strony	Strona {0}
Wyrównanie nagłówka strony	WYŚRODKOWANY
Czcionka nagłówka strony	[illegible]
Format nagłówka strony	Drukowanie {2} z {1}
Poprawa pozycji - rosnąca.	1.0
Zawiń linie	☐

Ustawienia Drukowania

Figure 3. Polish translation of NetBeans 6.0 Advanced Options window)

or breaking the project.

This leads to a second challenge: how to recruit new team members and keep them interested in the project. One or two team members usually volunteer to take on a leadership role, or are nominated by the other members. They are the ones who organize the work, create instructions and make sure all of the tasks are assigned. And they come to understand the importance of their role in keeping a team connected, encouraged and interested in contributing something for which there is no tangible reward.

How to ensure an acceptable level of product quality is a third challenge, especially when there is more than a handful of team members or a high turnover of contributors. Writing style and terminology usage need to be consistently maintained across all components and modules of the product. One way to achieve this is by using translation editing tools, such as OmegaT (see the **box** "Maximizing translation reuse"), and sharing the translation memory output among all team members.

"I am a free software fan, and NetBeans offers many qualities I was looking for in an IDE: it's intuitive, productive, extensible, has a well-organized community behind it, and the support of a company that likes open source. I've been using NetBeans for years and want it to be the best and most popular IDE in Burkina Faso."

Nacer Adamou Saidou,
Network and Systems Engineer

Additionally, Sun shares any existing style guides and term databases with the community. The contributors need to be in constant dialogue, so they use mailing lists in their language or the NetBeans collaboration module to discuss discrepancies and doubts. Team activities, progress, and decisions are documented on team wiki pages. And those with more linguistic and technical expertise take on the role of reviewing the translations of other less-experienced contributors, or provide linguistic consulting.

How to participate

If you are inspired to get involved in localizing the NetBeans IDE, the TranslatedFiles project home page explains how to get started. The first step is to subscribe to the project mailing list, introduce yourself to the other contributors, and check if a team for your language already exists. (Before you are granted commit rights to the workspace, you will need to sign and submit a Contributor Agreement.)

If translations are available from a previous version, instructions outline how to migrate these from one version to the next, so that an open-source trans-

Maximizing translation reuse

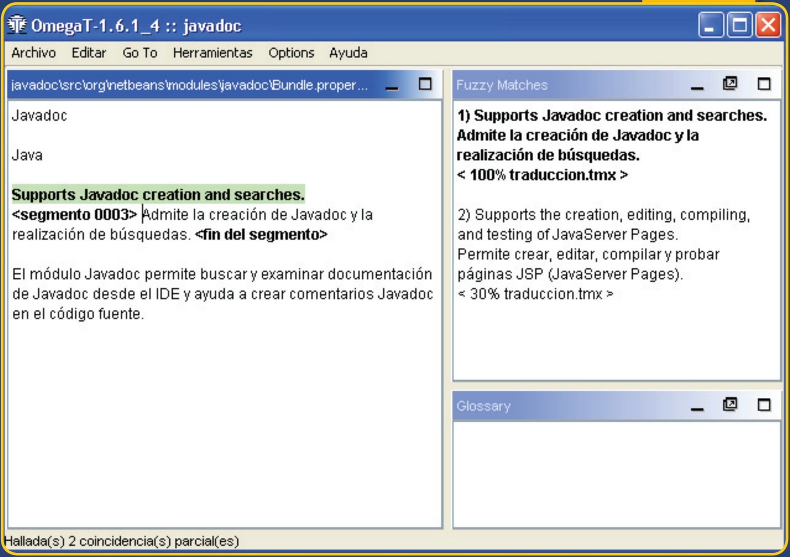
Imagine that each time there is a new version of NetBeans you have to translate all the messages and user interface from scratch. That doesn't seem like such a problem until you realize that you've seen many of these messages before, or perhaps that other members of your translation team are coming across the same set of messages as you are – and possibly making different translations of the same items. The good news is that the old translations can be reused or shared in new versions.

One solution is to make use of translation memory tools. These tools store the bilingual versions of translated segments and allow you to retrieve them later when translating software messages or documentation that contain the same or similar segments as previous versions. A segment is a unit of text divided at the sentence level; most tools have the option of adjusting segments to the phrase or paragraph level too, depending on the requirements of the project.

One of the most popular translation editing tools for open-source translation projects is OmegaT (*omegat.org*), an open-source application written in Java. As the translator works, the tool will search the database for matching pairs of segments. When a match is found the translator is offered the option of accepting or rejecting one or more previous translations. An example can be seen in **Figure B1**. An English segment is highlighted in green on the left side. The tool has found two possible matches in the Spanish translation database. One match is exact or 100%. The second is a fuzzy match, or 30% match between the old and new English strings.

There are many benefits of using translation memory tools. If substantial matches are found from previous translations, then less time is required to localize the product. This is because translators only have to handle a segment once, assuming it is repeated elsewhere within the same product, across other products, or in subsequent releases. Moreover, the translators do not need to be familiar with a wide range of file formats, nor do they need to own the software for processing them. For example, you might need a number of different editors or word processors to be able to process formats such as HTML, DOC, XML, etc. The translation memory tools convert these to a standard format that can be easily read by a number of translation editing tools. Finally, a high level of quality and consistency is maintained when translators can share and improve on terminology across many related projects.

B1


OmegaT-1.6.1_4 :: javadoc
Archivo Editar Go To Herramientas Options Ayuda
javadoc\src\org\netbeans\modules\javadoc\Bundle.proper...
Fuzzy Matches
Glossary

Javadoc

Java

Supports Javadoc creation and searches.
<segmento 0003> Admite la creación de Javadoc y la realización de búsquedas. <fin del segmento>

El módulo Javadoc permite buscar y examinar documentación de Javadoc desde el IDE y ayuda a crear comentarios Javadoc en el código fuente.

1) Supports Javadoc creation and searches.
Admite la creación de Javadoc y la realización de búsquedas.
< 100% traduccion.tmx >

2) Supports the creation, editing, compiling, and testing of JavaServer Pages.
Permite crear, editar, compilar y probar páginas JSP (JavaServer Pages).
< 30% traduccion.tmx >

Hallada(s) 2 coincidencia(s) parcial(es)

netbeans.org/community/articles/es-de-localization.html

Article about the Spanish and German teams and NetBeans release 5.5.1

Figure B1. Using the OmegaT translation editor (in Spanish) to check for matches from previous translations

lation editor can be used to leverage and preserve the reusable translation memory. The Sun engineers prepare the localization kits, which consist of the localizable elements – *.properties*, *.xml* and *.html* files. Once the translations are completed, reviewed by others on the team and committed to the CVS, it is time to test and verify the localized strings. The Sun Build Engineering team will eventually make weekly development builds available on the netbeans.org site for review and testing purposes. It's also possible to build a local version of the localized IDE when official builds are not yet available (see the **box** "How to build a localized NetBeans 6.0").

There are other ways to contribute translations, especially for those who have a particular interest in website content or product documentation. The netbeans.org community page for localization (*netbeans.org/community/contribute/localise.html*) lists a variety of opportunities for submitting translations. The most visible pages to contributors are the community page translations (*netbeans.org/community/contribute/localisation-status.html*). Contributors are continually sought to help keep the twenty-eight languages in sync with the regular updates of the English release.

"Translating NetBeans to Bahasa Indonesian, will help people learn to use the IDE more easily in our country. There is increasing attention to NetBeans in Indonesia. In a developing country with more than 230 million population, NetBeans' contribution is important for us to acquire new technology without great cost."

Ibrahim F. Burhan, Information Technologist for an Indonesian mining company and NetBeans user since version 3.5

Additionally, the community docs project (*wiki.netbeans.org/wiki/view/CommunityDocs*) offers the possibility to contribute content in English or other languages. Community members have submitted translations of tutorials and other NetBeans products and release documentation. They have also created original flash demos, white papers, and articles in various languages.

Three NetBeans portals exist currently in languages other than English: *ja.netbeans.org*, *zh-cn.netbeans.org* and *fr.netbeans.org*. You can contribute original or translated content to these sites to spread knowledge about NetBeans to developers in countries or communities where these languages are spoken.

All of these projects are ways to contribute to NetBeans as the IDE of choice worldwide. It is a way to share your skills and expertise with others, and help reach developers who might not be aware of the great features offered by NetBeans. The benefits of contributing to the TranslatedFiles translation project are numerous. You can make the IDE more accessible to international developers, and, in the process, enjoy the camaraderie of sharing knowledge, experiences, and social interaction with other people who love NetBeans and believe in the open source ideology.

netbeans.org/community/articles/translations.html

Article about NetBeans Russian community contributor and TranslatedFiles founder, Maxym Mykhalchuk

Building a localized NetBeans 6.0

Once the translations of the message files are completed, they will need to be committed to the CVS and then integrated into the build environment. At this point, it is important to validate the quality, accuracy, and completeness of the translated strings, by reviewing them in the context of the actual product. Development builds for testing might not be available on netbeans.org until later in the development cycle, but contributors can still build a local version with the translations, as follows.

Before beginning, make sure that Apache Ant 1.7, JDK 1.5 and a CVS client are installed on your local machine. Then set up your environment and checkout the NetBeans source files from the CVS (see *wiki.netbeans.org/wiki/view/WorkingWithNetBeansSources* for details). We'll use BUILD_HOME to refer to the top-level directory.

You must build the entire IDE before you can build the localized version:

```
% cd BUILD_HOME/nbbuild
% ant build-nozip
```

The NetBeans IDE will be built under *nbbuild/netbeans*. You can start NetBeans from *nbbuild/netbeans/bin/netbeans*.

Next, you need to checkout *translatedfiles/src* from CVS (the checkout steps in the Wiki mentioned above will not grab them):

```
% cd BUILD_HOME
% cvs co -r release60 translatedfiles/src
```

If the translated message files (HTML, XML, properties, etc.) have been integrated already into the *translatedfiles/src* directory of the CVS, you can use them as they are. If you prefer, however, to validate and check your translations before integrating them in to the CVS, then make sure to put them into the *translatedfiles/src* directory on your local machine where you will do the local build. The next step is to build the localized JAR files:

```
% cd BUILD_HOME/nbbuild
% ant -Dlocales=<lang> build-nozip-ml
```

Here is an example using "ja" (for Japanese).

```
% ant -Dlocales=ja build-nozip-ml
```

After the build process is done, the *_<lang>.jar files will be created under the *nbbuild/netbeans* directory. Now it's time to start the localized version of NetBeans:

```
% BUILD_HOME/nbbuild/netbeans/bin/netbeans
```

If your operating system's locale is different from the locale in which you will run NetBeans, use the *--locale* option to specify the language:

```
% BUILD_HOME/nbbuild/netbeans/bin/netbeans
    --locale <language[:country[:variant]]>
```

Here is an example using Brazilian Portuguese:

```
% BUILD_HOME/nbbuild/netbeans/bin/netbeans --locale pt:BR
```

You can now proceed to test the translations in the localized build environment. If there are multiple team members for your language, you can distribute the *_<lang>.jar files to the other team members. They'll just need to unzip these files into the NetBeans installation directory.

by Masaki Katakai

netbeans.org/community/articles/brazil55-release.html

About the Brazil team and NetBeans release 5.5

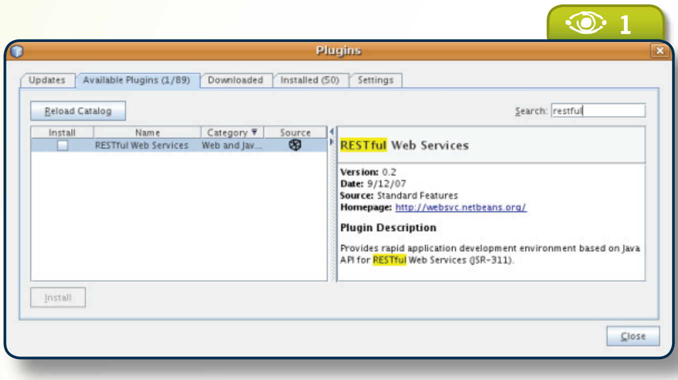
Janice Campbell (*janice.campbell@sun.com*) is a Globalization Program Manager at Sun Microsystems, Inc. For the last year and a half, she worked with a Sun team of localizers in Japan, China and the Czech Republic, and with an ever-changing and growing team of community contributors from all parts of the globe.

Building RESTful Web Services

in NetBeans 6.0

Rapid code generation based on patterns and JPA entities, easier testing with Test Client, invoking third-party services with RESTful components and building client applications using generated JavaScript client libraries

Peter Liu



Plugins

Updates | Available Plugins (1/89) | Downloaded | Installed (50) | Settings

Reload Catalog

Search: restful

Install	Name	Category	Source
☐	RESTful Web Services	Web and Jav...	

RESTful Web Services

Version: 0.2
Date: 9/12/07
Source: Standard Features
Homepage: http://websvc.netbeans.org/

Plugin Description

Provides rapid application development environment based on Java API for RESTful Web Services (JSR-311).

Install

Close

Figure 1. Downloading the RESTful Web Services Plugin from the Plugin Manager

Representational State Transfer (REST), you will recall, is an increasingly popular architectural style for distributed hypermedia systems such as the Web. Central to the RESTful architecture style is the concept of resources identified by universal resource identifiers (URIs). Resources can be manipulated using a standard interface, such as HTTP, and information is exchanged using representations of these resources. Building web services using the RESTful approach is emerging as a popular alternative to using SOAP-based technologies for deploying services on the Internet, due to its lightweight nature.

NetBeans 6.0, with the addition of a specialized plugin, supports the rapid development of RESTful web services using JSR-311 (Java API for RESTful Web Services – JAX-RS) and Jersey, the reference implementation for JAX-RS. The IDE supports building and testing services, as well as creating client applications that access these services, and generating code for invoking web services (not only RESTful but also SOAP-based). The following are the RESTful features provided:

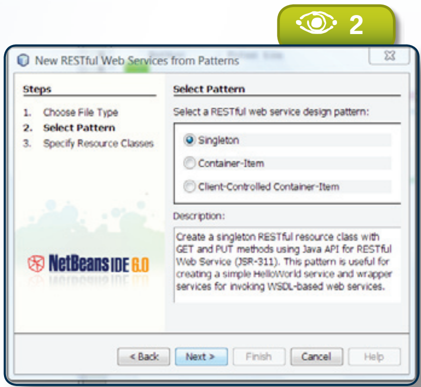
- Rapid creation of RESTful web services from Java Persistence API entity classes and patterns.
- Fast code generation for invoking web services such as Google Maps, Yahoo News Search, and StrikeIron services, by drag-and-dropping components from the RESTful component palette.
- Generation of JavaScript client stubs from RESTful web services for building client applications.
- Test client generation for testing RESTful web services.
- A logical view for easy navigation of RESTful web service implementation classes in your project.

In this article, we will outline the steps to use these features to create RESTful applications.

Getting the RESTful Web Services Plugin

Since we are tracking JSR-311 which is still an unfinished specification at the time of writing, the RESTful Web Services plugin is only available via the NetBeans 6.0 Plugin Manager from the Update Center. To download it, you need to have installed NetBeans 6.0 in the *Web and Java EE* or *Complete* releases. Once in the IDE, go to *Tools>Plugins* and install the plugin, as shown in **Figure 1**. You'll then be ready to explore the various features available for developing and testing your RESTful applications.

The RESTful Web Services Plugin comes bundled with the latest Jersey libraries, so you do not need to install the Jersey runtime in your application server in order to run your applications.



New RESTful Web Services from Patterns

Steps
1. Choose File Type
2. **Select Pattern**
3. Specify Resource Classes

Select Pattern

Select a RESTful web service design pattern:

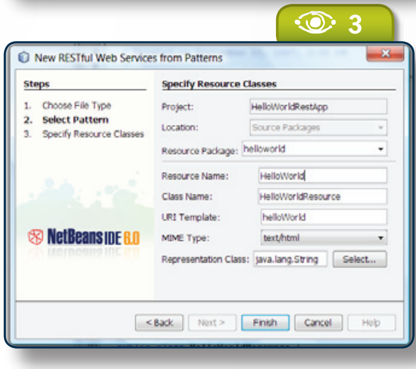
(●) Singleton
( ) Container-Item
( ) Client-Controlled Container-Item

Description:

Create a singleton RESTful resource class with GET and PUT methods using Java API for RESTful Web Service (JSR-311). This pattern is useful for creating a simple HelloWorld service and wrapper services for invoking WSDL-based web services.

< Back | Next > | Finish | Cancel | Help

Figure 2. Pattern Selection Panel



New RESTful Web Services from Patterns

Steps
1. Choose File Type
2. **Select Pattern**
3. Specify Resource Classes

Specify Resource Classes

Project: HelloWorldRestApp
Location: Source Packages
Resource Package: helloworld
Resource Name: HelloWorld
Class Name: HelloWorldResource
URI Template: helloWorld
MIME Type: text/html
Representation Class: java.lang.String

Select...

< Back | Next > | Finish | Cancel | Help

Figure 3. Setting up resource classes

A first RESTful Web Service

Start by creating a Web project with the type *Web Application* and default settings. From the Web project node, choose *New>RESTful Web Services from Patterns*. **Figure 2** shows the first panel of the wizard. As you'll see, three patterns are currently supported: Singleton, Container-Item and Client-Controlled Container-Item (see more about these in the "RESTful patterns" box). We will use the Singleton pattern to create a "Hello World" service. Click on the *Next* button to go to the panel shown in **Figure 3**.

In this panel, you can specify the package and resource names, and the MIME type for your resource. Enter "helloworld" in the *Resource Package* text field, and in *Resource Name* enter "HelloWorld". The *Class Name* and *URI Template* fields will default to "HelloWorldResource" and "helloWorld", respectively. Select *text/html* for the MIME Type, and click *Finish* to generate the code.

After the **HelloWorldResource** class is generated, the RESTful Web Services logical view will appear, as shown in **Figure 4**. Double click on the **getHtml()** method node, and the IDE will take you to the method in the Java editor. Modify the method as shown in **Listing 1**, to return an HTML document displaying "Hello World!".

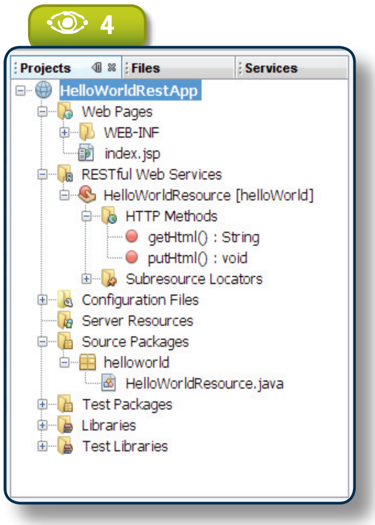


Figure 4. RESTful Web Services Logical View

Taking a closer look at the generated *HelloWorldResource.java* class shown in **Listing 1**, notice the **@UriTemplate("/helloWorld")** annotation on the class definition. This annotation is what determines the URI for the resource. The **@HttpMethod("GET")** and **@ProduceMime("text/html")** annotation on the **getHtml()** method indicates to the Jersey runtime that this method should be called when an HTTP GET request is sent to the resource with *text/html* as the desired MIME type.

The HelloWorld project we created in this section is one of the sample applications bundled with the plugin. You can create the sample by choosing *File>New Project>Samples>RESTful Web Services>Hello World.*

We've completed our minimal service. In the next section we demonstrate how to test the service using the *Test Client* facility.

Testing the service

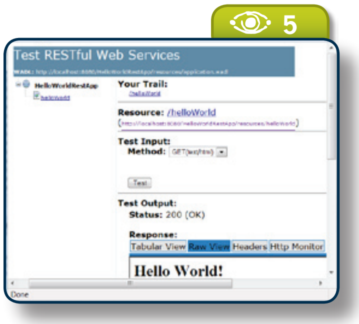
To test the HelloWorld service, right click on the project node and select *Test RESTful Web Services*. This action will deploy the application and bring up the test client in your browser, as shown in **Figure 5**. To test the service, click on the *helloWorld* node on the

Listing 1. HelloWorldResource.java

```
@UriTemplate("helloWorld")
public class HelloWorldResource {
  ...
  @HttpMethod("GET")
  @ProduceMime("text/html")
  public String getHtml() {
   return "
     <html>
       <body>
         <h1>Hello World!</h1>
       </body>
     </html>";
  }
  ...
}
```

More on REST

RESTful web services are much simpler than the traditional SOAP-based ones. There is of course no SOAP and no WSDL, and definitely no WS-* standards – just simple message exchange over HTTP using any format you want: XML, JSON, HTML, etc. RESTful web services are really just a collection of web resources identifiable by URIs, which can be manipulated by a small number of operations – GET, PUT, POST and DELETE (corresponding to commons actions you can specify in a HTTP header). Because of this, "resource" is used throughout the article to mean a RESTful web service.


Test RESTful Web Services

WADL: http://localhost:8080/HelloWorldRestApp/resources/application.wadl

HelloWorldRestApp
helloWorld

Your Trail: /helloWorld

Resource: /helloWorld (http://localhost:8080/HelloWorldRestApp/resources/helloWorld)

Test Input:
Method: GET(text/html)

Test Output:
Status: 200 (OK)

Response:

Hello World!

Figure 5. Test Client for our minimal web service

left-hand panel and click *Test*. Switch to the *Raw View* to see the HTML document. The *Tabular View* extracts all the URIs in the result document, and displays them as links. We will show an example of this later in the article. The *Headers* view shows all the information in the HTTP headers, and the *HTTP Monitor*, as expected, displays HTTP request and response messages.

! The Test Client generates its content based on the generated WADL (Web Application Description Language) file which is a description for all the RESTful resources in your project.

Generating RESTful web services from a database

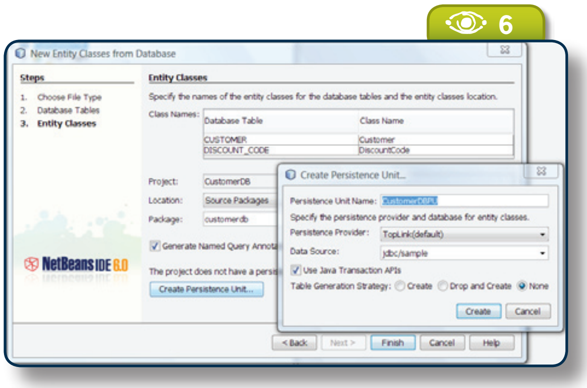
In addition to generating code from RESTful patterns, you can generate a fully functional RESTful application from a database, leveraging the JPA support in NetBeans. To do this, we will need to generate the JPA entity classes from a database in the web project.

First create a web project called "CustomerDB", keeping GlassFish V2 as the default server. From the project node, choose *New>Entity Classes from Database* and select *jdbc/sample* for Data Source from the drop-down list. A list of tables will appear in the *Available Tables* column. Select the **CUSTOMER** and **DISCOUNT_CODE** tables and click *Add* to include them in the *Selected Tables* column. Click *Next* to go the panel shown in **Figure 6**. Enter **customerdb** for *Package*. Next, create a persistence unit by clicking on the *Create Persistent Unit* button (you can leave everything as default in the dialog and click *Create*). Click *Finish* to generate the entity classes. After generation, you should see two Java classes, **Customer** and **DiscountCode**, in the **customerdb** package.

You are now ready to generate RESTful web services from the entity classes. Right click on the **customerdb** package node and choose *New>RESTful Web Services from Entity Classes* to bring up the New RESTful Web Services from Entity Classes wizard as shown in **Figure 7**. In this panel, simply click on *Add All* to include both entity classes for code generation. Click *Next* to go the next panel, shown in **Figure 8**.

More on Jersey

Jersey is more than just the reference implementation for JAX-RS. It's an open source project meant to foster a community of users and developers and is developed to be of production quality. On top of the support for the JAX-RS API, Jersey provides APIs and extension points. For example, you can extend Jersey to support additional languages such as Groovy and Scala, and to support different MIME types beyond XML and JSON.


New Entity Classes from Database

Steps
1. Choose File Type
2. Database Tables
3. **Entity Classes**

Entity Classes

Specify the names of the entity classes for the database tables and the entity classes location.

Class Names:

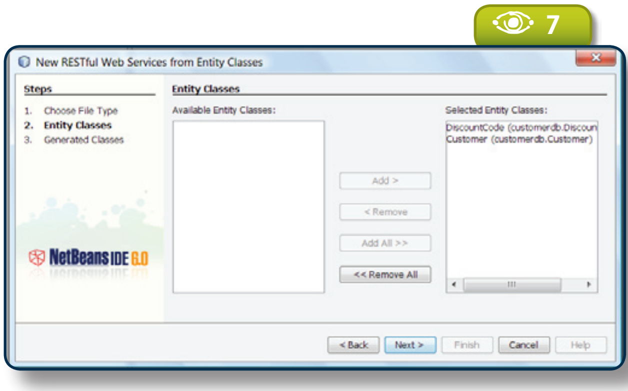
Database Table	Class Name
CUSTOMER	Customer
DISCOUNT_CODE	DiscountCode

Project: CustomerDB
Location: Source Packages
Package: customerdb
[✓] Generate Named Query Annota
The project does not have a persis

Create Persistence Unit...

Persistence Unit Name: CustomerDBPU
Specify the persistence provider and database for entity classes.
Persistence Provider: TopLink(default)
Data Source: jdbc/sample
[✓] Use Java Transaction APIs
Table Generation Strategy: Create / Drop and Create / [✓ None]

Figure 6. Entity Class Setup Panel


New RESTful Web Services from Entity Classes

Steps
1. Choose File Type
2. **Entity Classes**
3. Generated Classes

Entity Classes

Available Entity Classes:

Selected Entity Classes:
DiscountCode (customerdb.Discoun
Customer (customerdb.Customer)

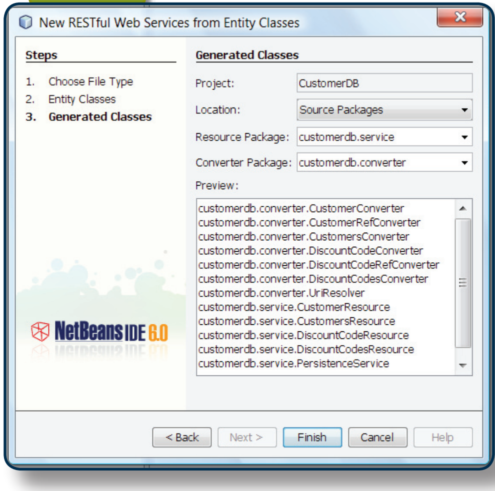
Figure 7. Entity Class Selection Panel

This panel displays a preview of the classes to be generated. The IDE uses the Container-Item pattern to generate the resource classes. For example, for the **Customer** entity class, the IDE generates a container resource called **CustomersResource** and an item resource called **CustomerResource**. Also, for each resource class, NetBeans creates a converter class which is used for generating the resource representation from the corresponding entity instance, e.g. **CustomersConverter** and **CustomerConverter**. Note that there are additional converter classes, e.g. **CustomerRefConverter**, for representing relationships.

Click *Finish* to generate all the classes indicated in the *Preview* area. You can test the generated RESTful web service by following the steps outlined in the previous section. **Figure 9** shows an example output. The Tabular View displays a listing of all the URIs embedded in the returned document; you can navigate to them by clicking on each link.

Taking a closer look at the **CustomersResource** class (shown in **Listing 2**), notice that the **getCustomerResource()** method has a **@UriTemplate("{customerId}")** annotation. This method is called a "sub-resource locator", because it is used to return an instance of the **CustomerResource** class to handle the HTTP request, after the Jersey runtime matches the **customerId** in the URI to the pattern specified in **@UriTemplate**. For example, the URI */customers/1* will be matched to this method because */customers* is first matched to **@UriTemplate("/customers")** in the **CustomersResource** class and "1" will match the **@UriTemplate("{customerId}")** annotation for the **getCustomerResource()** method.

Also notice in the **get()** method, also shown in **Listing 2**, that there are two parameters, **start** and **max** with **@QueryParam** annotations on them. These annotations are used by the runtime to inject values of the query parameters specified in a URI into the method parameters. For example, */customers?max=20* would cause the **max** parameter for the **get()** method to be set to 20 at runtime. The **@DefaultValue** annotation is used to specify default values if no query parameters are specified in the URI. The runtime will convert the value to the type specified by the method parameter.



New RESTful Web Services from Entity Classes

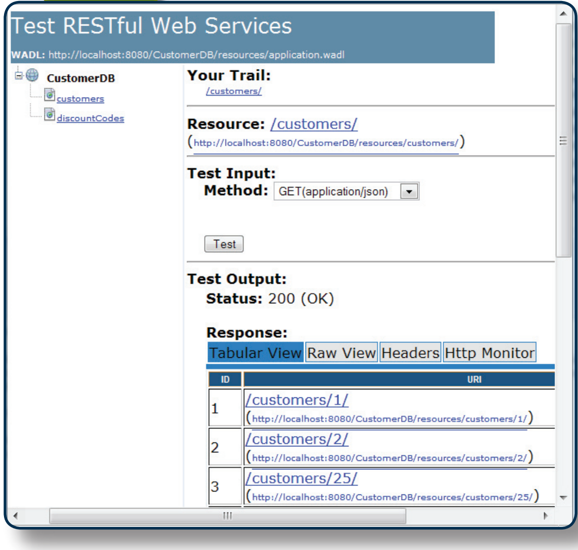
Steps
1. Choose File Type
2. Entity Classes
3. **Generated Classes**

Generated Classes

Project: CustomerDB
Location: Source Packages
Resource Package: customerdb.service
Converter Package: customerdb.converter
Preview:

```
customerdb.converter.CustomerConverter
customerdb.converter.CustomerRefConverter
customerdb.converter.CustomersConverter
customerdb.converter.DiscountCodeConverter
customerdb.converter.DiscountCodeRefConverter
customerdb.converter.DiscountCodesConverter
customerdb.converter.UriResolver
customerdb.service.CustomerResource
customerdb.service.CustomersResource
customerdb.service.DiscountCodeResource
customerdb.service.DiscountCodesResource
customerdb.service.PersistenceService
```

Figure 8. Resource Class Setup Panel



Test RESTful Web Services

WADL: http://localhost:8080/CustomerDB/resources/application.wadl

CustomerDB
- customers
- discountCodes

Your Trail:
/customers/

Resource: /customers/
(http://localhost:8080/CustomerDB/resources/customers/)

Test Input:
Method: GET(application/json)

Test Output:
Status: 200 (OK)

Response:

ID	URI
1	/customers/1/ (http://localhost:8080/CustomerDB/resources/customers/1/)
2	/customers/2/ (http://localhost:8080/CustomerDB/resources/customers/2/)
3	/customers/25/ (http://localhost:8080/CustomerDB/resources/customers/25/)

Figure 9. Testing RESTful Web Services Generated from Entity Classes

Listing 2. CustomersResource.java

```
@UriTemplate("/customers/")
public class CustomersResource {
  ...
  @HttpMethod("GET")
  @ProduceMime({"application/xml", "application/json"})
  public CustomersConverter get(
    @QueryParam("start") @DefaultValue("0") int start,
    @QueryParam("max") @DefaultValue("10") int max) {
      ...
    }
  ...
  @UriTemplate("{customerId}/")
  public CustomerResource getCustomerResource() {
    return new CustomerResource(context);
  }
  ...
}
```

Now take a look at the **CustomerResource** class shown in **Listing 3**. Notice the **@UriParam("customerId")** annotation for the **id** parameter on the **get()** method. What this annotation does is tell the Jersey runtime to extract the **customerId** from the URI and inject its value into the **id** parameter. The runtime will convert the **customerId** to an **Integer**, as specified by the parameter type.

The generated RESTful web services support both XML and JSON MIME types. You can specify which MIME type to use by selecting the appropriate one from the *Method* drop-down list.

RESTful components

In addition to generation of RESTful web services from patterns and JPA entity classes, the plugin supports generating code to access other web services – both RESTful and SOAP-based. This is done using the RESTful components available in the component palette. To bring up the palette with the relevant items, simply open up a RESTful resource class, e.g. **CustomerResource**. **Figure 10** shows the RESTful components currently available. (If the palette does not come up, you can open it with *Window|Palette* or *CTRL+Shift+8*.)

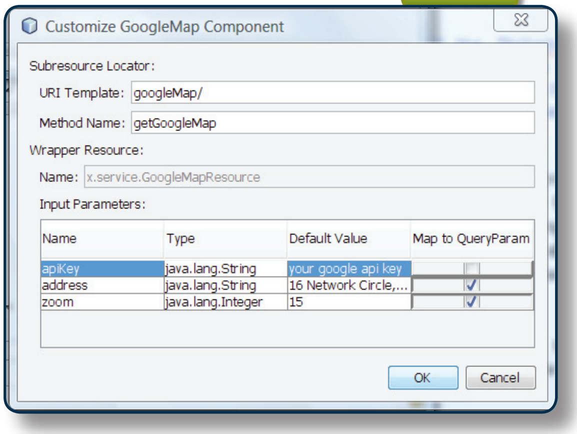
! Google Map and Yahoo News Search components are both REST-based; all StrikeIron components are SOAP-based. Currently, you can add components to the palette by creating modules with metadata information in a specific format (the format for such modules is beyond the scope of this article, however).

To demonstrate how to use these components, drag and drop the Google Map component into your **CustomerResource** class. A dialog will appear (see **Figure 11**), where you can customize the Google Map component. For the *apiKey*, enter the key which you obtained from Google (by visiting *google.com/apis/maps/signup.html*). You should also uncheck the *Map to QueryParam* checkbox for the *apiKey*, so it does not appear as a query parameter.

Click *Ok* and the plugin will generate a sub-resource locator called **getGoogleMap()** in the **CustomerResource** class. The plugin also generates a **GoogleMapResource** class with pre-generated code for invoking the Google Map API.

To have the Google Map keyed to the address of the customer, you need to modify the code as shown in **Listing 4**. Run *Test RESTful Web Services* again to test the modified application. **Figure 12** shows the Test Client page. Notice that a new URI, */customers/1/googleMap* is added to the **Customer** resource. If you click on the link, you'll see the map shown in **Figure 13**.

11



Subresource Locator:
URI Template: googleMap/
Method Name: getGoogleMap
Wrapper Resource:
Name: x.service.GoogleMapResource
Input Parameters:

Name	Type	Default Value	Map to QueryParam
apiKey	java.lang.String	your google api key	☐
address	java.lang.String	16 Network Circle,...	✓
zoom	java.lang.Integer	15	✓

Figure 11. Customizing the GoogleMap Component

10

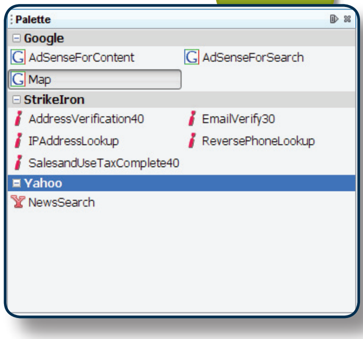
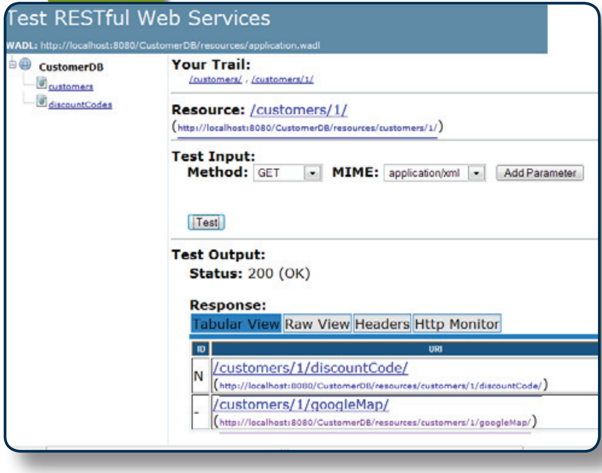


Figure 10. The REST Component Palette

Listing 3. CustomerResource.java

```
public class CustomerResource {
  ...
  @HttpMethod("GET")
  @ProduceMime({"application/xml", "application/json"})
  public CustomerConverter get(
    @UriParam("customerId") Integer id) {
    ...
  }
  ...
}
```



Test RESTful Web Services

WADL: http://localhost:8080/CustomerDB/resources/application.wadl

CustomerDB
- customers
- discountCodes

Your Trail:
/customers/ , /customers/1/

Resource: /customers/1/
(http://localhost:8080/CustomerDB/resources/customers/1/)

Test Input:
Method: GET **MIME:** application/xml Add Parameter

Test

Test Output:
Status: 200 (OK)

Response:
Tabular View | Raw View | Headers | Http Monitor

ID	URI
N	/customers/1/discountCode/ (http://localhost:8080/CustomerDB/resources/customers/1/discountCode/)
-	/customers/1/googleMap/ (http://localhost:8080/CustomerDB/resources/customers/1/googleMap/)

Figure 12. Testing the GoogleMap Resource

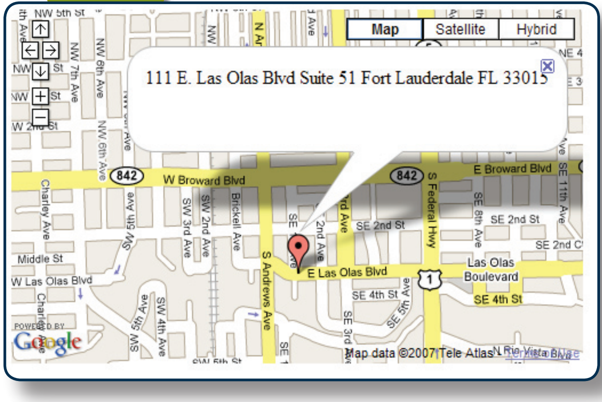
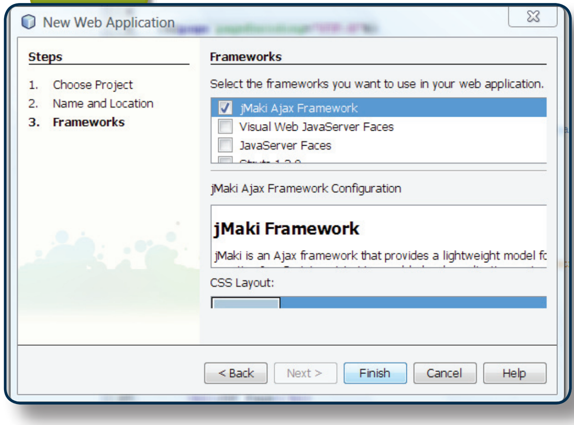


Figure 13. Google Map for Customer 1



New Web Application

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

Frameworks

Select the frameworks you want to use in your web application.

- [✓] jMaki Ajax Framework
- [ ] Visual Web JavaServer Faces
- [ ] JavaServer Faces

jMaki Ajax Framework Configuration

jMaki Framework

jMaki is an Ajax framework that provides a lightweight model fc

CSS Layout:

< Back | Next > | Finish | Cancel | Help

Figure 14. Adding the jMaki Framework

Client library generation

To facilitate building client applications that access RESTful web services, the plugin supports generation of client libraries written in JavaScript. The generated libraries can contain three layers of abstraction. At the bottom layer is plain JavaScript code which can be consumed by JavaScript clients. The next layer provides sup-

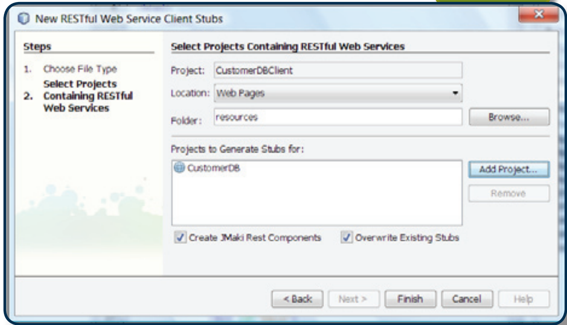
REST Patterns

The RESTful plugin currently supports three patterns. The **Singleton** pattern is useful for creating RESTful wrapper services around other services such as SOAP-based web services. For example, you can use this pattern to REST-enable existing SOAP-based services. This pattern has many additional uses.

The **Container-Item** pattern is typically used for representing resources that have relationships, such as a database. We use this pattern to generate code from JPA entity classes.

This "plain" Container-Item pattern lets you create an item resource using the POST method. The URI for the newly created item resource is determined by the container resource.

The **Client-Controlled Container-Item** pattern is a slight variation. The difference is that there is no POST method on the container resource class for creating item resources. Instead, item resources are created using PUT on the item resource class. It's called Client-Controlled because the URI for the item resource is determined by the client and not by the container resource. Amazon's Simple Storage Service (S3) uses the Client-Controlled Container-Item pattern.



New RESTful Web Service Client Stubs

Steps
1. Choose File Type
2. **Select Projects Containing RESTful Web Services**

Select Projects Containing RESTful Web Services

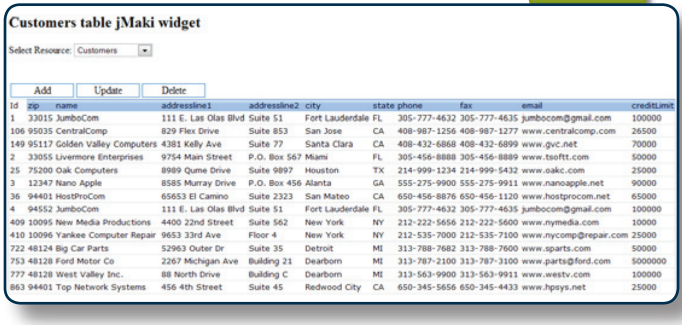
Project: CustomerDBClient
Location: Web Pages
Folder: resources
Browse...

Projects to Generate Stubs for:
CustomerDB
Add Project... | Remove

[✓] Create JMaki Rest Components [✓] Overwrite Existing Stubs

< Back | Next > | Finish | Cancel | Help

Figure 15. New RESful Web Service Client Stubs Wizard



Customers table jMaki widget

Select Resource: Customers

Add Update Delete

Id	zip	name	addressline1	addressline2	city	state	phone	fax	email	creditLimit
1	33015	JumboCom	111 E. Las Olas Blvd	Suite 51	Fort Lauderdale	FL	305-777-4632	305-777-4635	jumbocom@gmail.com	100000
106	95035	CentralComp	829 Flex Drive	Suite 853	San Jose	CA	408-987-1256	408-987-1277	www.centralcomp.com	26500
149	95117	Golden Valley Computers	4381 Kelly Ave	Suite 77	Santa Clara	CA	408-432-6868	408-432-6899	www.gvc.net	70000
2	33055	Livermore Enterprises	9754 Main Street	P.O. Box 567	Miami	FL	305-456-8888	305-456-8889	www.tsoftt.com	50000
25	75200	Oak Computers	8989 Qume Drive	Suite 9897	Houston	TX	214-999-1234	214-999-5432	www.oakc.com	25000
3	12347	Nano Apple	8585 Murray Drive	P.O. Box 456	Alanta	GA	555-275-9900	555-275-9911	www.nanoapple.net	90000
36	94401	HostProCom	65653 El Camino	Suite 2323	San Mateo	CA	650-456-8876	650-456-1120	www.hostprocom.net	65000
4	94552	JumboCom	111 E. Las Olas Blvd	Suite 51	Fort Lauderdale	FL	305-777-4632	305-777-4635	jumbocom@gmail.com	100000
409	10095	New Media Productions	4400 22nd Street	Suite 562	New York	NY	212-222-5656	212-222-5600	www.nymedia.com	10000
410	10096	Yankee Computer Repair	9653 33rd Ave	Floor 4	New York	NY	212-535-7000	212-535-7100	www.nycomp@repair.com	25000
722	48124	Big Car Parts	52963 Outer Dr	Suite 35	Detroit	MI	313-788-7682	313-788-7600	www.sparts.com	50000
753	48128	Ford Motor Co	2267 Michigan Ave	Building 21	Dearborn	MI	313-787-2100	313-787-3100	www.parts@ford.com	5000000
777	48128	West Valley Inc.	88 North Drive	Building C	Dearborn	MI	313-563-9900	313-563-9911	www.westv.com	100000
863	94401	Top Network Systems	456 4th Street	Suite 45	Redwood City	CA	650-345-5656	650-345-4433	www.hpsys.net	25000

Figure 16. Example using jMaki

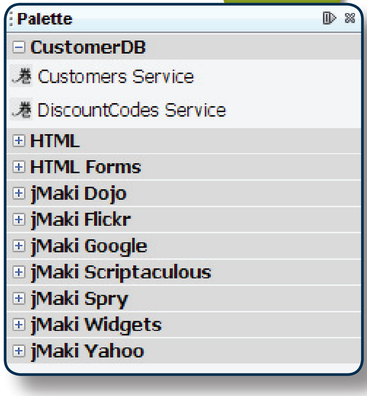


Figure 17. jMaki palette

port for the Dojo Ajax toolkit. The generated code is organized in a structure understood by Dojo and contains code for the Dojo store and widget abstractions. On the top is the jMaki layer, which is a wrapper around the Dojo layer.

In order to generate all three layers, you need to first install the jMaki plugin available in the download area of the *ajax.dev.java.net* website. This plugin will automatically install the Dojo libraries so you do not need to install these separately. (If you do not install the jMaki plugin, you'll only be able to generate plain JavaScript code.)

To create the client library, you need to first create a web project, say "CustomerDBClient". Also, make sure you add the jMaki framework as shown in **Figure 14**. Next, right click on the project node and invoke *New>RESTful Web Service Client Stubs*, to invoke the wizard shown in **Figure 15**.

Click on *Add Project* and select the *CustomerDB* project you created. Make sure the *Create JMaki Rest Components* checkbox is selected and click *Finish*. The plugin will generate all the necessary files in the project's web folder.

You can test the generated client library by running the *TestResourcesTable.jsp* page, in the *web/resources/dojo/rest* folder (simply right click on the file and choose *Run File*). **Figure 16** shows the result.

You can also install the generated client library as a jMaki component in the jMaki palette. In the root directory for the *CustomerDBClient* project, there should be a *CustomerDB.zip* file already created when you create the client library. You can see it in the *Files* view of the IDE. Next, choose *Tools>Palette>Add jMaki Library* and select *CustomerDB.zip*. The jMaki plugin will add the jMaki components contained in this file to the jMaki palette, as shown in **Figure 17**. Finally, to test the component, simply drag and drop it into a JSP file such as *index.jsp*, and run it from the IDE.

Listing 4. getGoogleMap() method

```
  @UriTemplate("googleMap/")
   public GoogleMapResource getGoogleMap(
      @UriParam("customerId") Integer id)
   {
     try {
       x.Customer entity = getEntity(id);
       String apiKey = null;
       String address = entity.getAddressline1() + " " +
           entity.getAddressline2() + " " +
           entity.getCity() + " " +
           entity.getState() + " " +
           entity.getZip();
       Integer zoom = null;
       return new GoogleMapResource(apiKey, address, zoom);
     } finally {
       PersistenceService.getInstance().close();
     }
   }
```

Conclusions

In this article, we gave you an introduction to the world of RESTful web services, and the current efforts by JAX-RS and Jersey to standardize on a Java API for building such web services. We also showcased the RESTful Web Service plugin for NetBeans 6.0, which provides end-to-end support for building complete RESTful web services.

Peter Liu (*peter.liu@sun.com*) is a staff engineer at Sun Microsystems, Inc. leading the RESTful Web Services tooling effort for NetBeans 6.0.

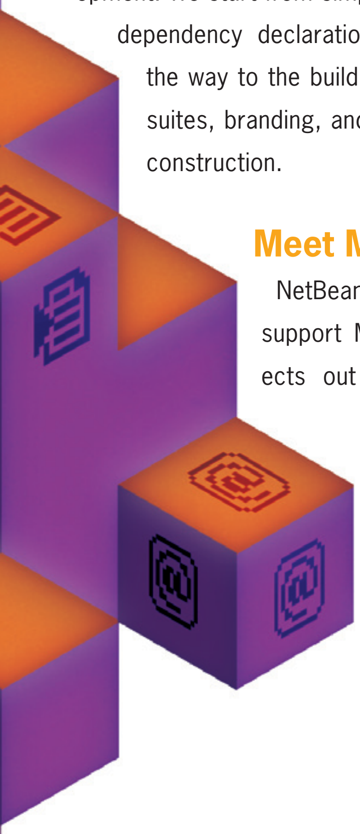
NetBeans Platform Development with Maven 2

Creating a Module Suite with Apache Maven and Mevenide – from basic Platform API features to JavaHelp support and branding

Emilian Bold

Apache Maven, you all know, is widely used as a build system and for many other activities. A great thing about Maven is that its "build script" is actually no script at all but a completely declarative configuration file called a POM (Project Object Model). Maven's design will look familiar to NetBeans Platform developers: it's basically constructed from a core "platform" supporting versioned plugins that can be automatically downloaded from a central repository.

This article will show that NetBeans is starting to have excellent Maven support, and how to use this as an alternative to the IDE's built-in Ant integration – for every aspect of NetBeans Platform development. We start from simple issues like dependency declaration and go all the way to the building of module suites, branding, and help module construction.



Meet Mevenide

NetBeans does not yet support Maven 2 projects out of the box. Luckily though, we have Mevenide, a certified NetBeans plugin that provides extensive Maven integration. You can use existing Maven projects directly from the IDE as Mevenide provides execution and debugging support, auto-completion for many Maven-specific files, and more. All projects created with Mevenide will be standard Maven 2 projects that can also be built with the command-line **mvn** command.

But if your projects will be standard Maven 2 projects, there's nothing actually forcing you to use the NetBeans IDE; so what's to gain as a Platform developer? Well, by standardizing on Maven, members of your team could use different IDEs or even plain text editors to do the development. In particular, you can build NetBeans Platform applications with whatever tools you prefer.

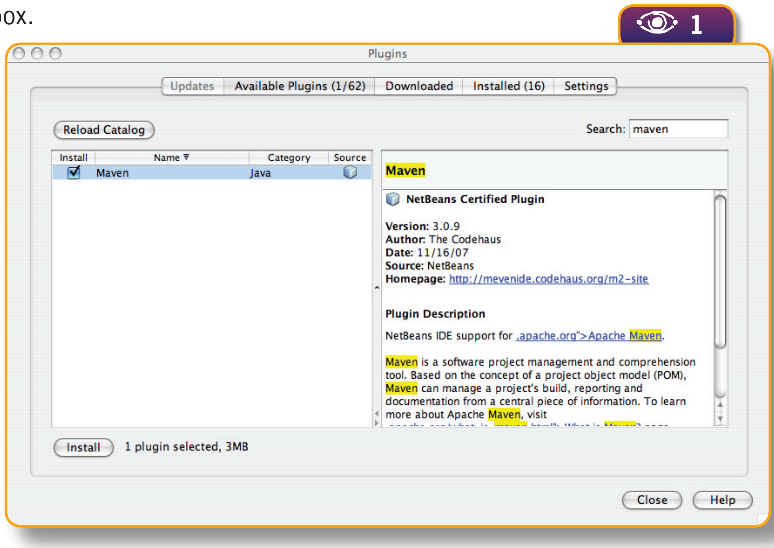
The downside to using Mevenide and Maven 2 projects is that, while you do get independence from the IDE and an arguably better build system than Ant, you lose some IDE integration. For example, some of the wizards are gone regardless of the project type. For Platform development in particular, you'll have to hand-edit some of the properties or XML files (the *layer.xml* file being the prime candidate). In some cases the loss of integration is partial; for example, the form editor will work but you won't be able to edit the layer using drag and drop.

All that said, keep an eye on the update center as the missing wizards are slowly coming to Maven-based projects.

Mevenide can be easily installed by selecting *Tools|Plugin*, choosing "Maven" from the list of plugins (see **Figure 1**), and going through the normal installation steps.

mojo.codehaus.org/nbm-maven-plugin/descriptor.html

Sample NBM Maven Plugin descriptors



Install	Name	Category	Source
✓	Maven	Java	

Maven

NetBeans Certified Plugin

Version: 3.0.9
Author: The Codehaus
Date: 11/16/07
Source: NetBeans
Homepage: http://mevenide.codehaus.org/m2-site

Plugin Description

NetBeans IDE support for .apache.org">Apache Maven.

Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. To learn more about Apache Maven, visit

1 plugin selected, 3MB

Figure 1. The Plugins window after manual selection

The first module

Let's start creating a Maven-based NetBeans module. The first steps are the same for any Maven project: select *File>New Project*, open the *Maven* category and choose *Maven Project*. We'll use the *Quickstart Archetype* (see **Figure 2**) for this module.

An Archetype is basically a project generator in the Maven world. It produces the initial folder layout and the files to build upon.

In the final step, we define the *Group Id*, *Artifact Id*, and *Version*, as well as the project name (see **Figure 3**). These pieces of information together identify each artifact generated and manipulated by Maven (including the project itself), and will go into the project's POM. The *Group Id* is basically a namespace – it's common practice to use company, domain or application names

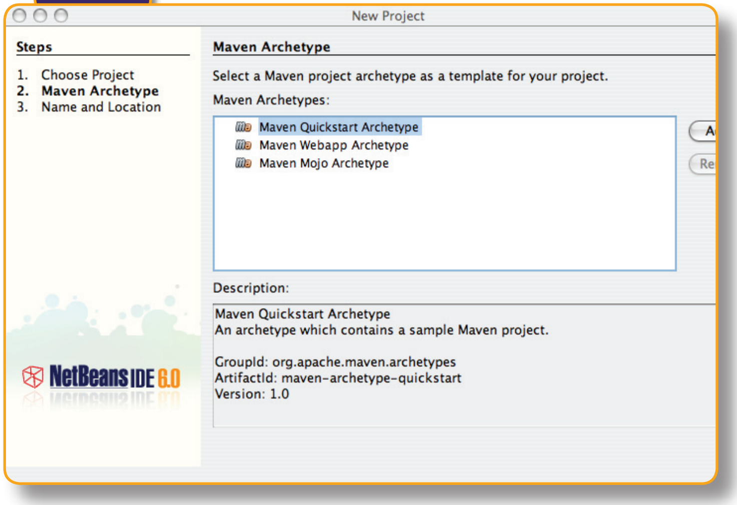


Figure 2. Selecting a Maven Archetype

New Project

Steps
1. Choose Project
2. **Maven Archetype**
3. Name and Location

Maven Archetype

Select a Maven project archetype as a template for your project.

Maven Archetypes:
- Maven Quickstart Archetype
- Maven Webapp Archetype
- Maven Mojo Archetype

Description:

Maven Quickstart Archetype
An archetype which contains a sample Maven project.

GroupId: org.apache.maven.archetypes
ArtifactId: maven-archetype-quickstart
Version: 1.0

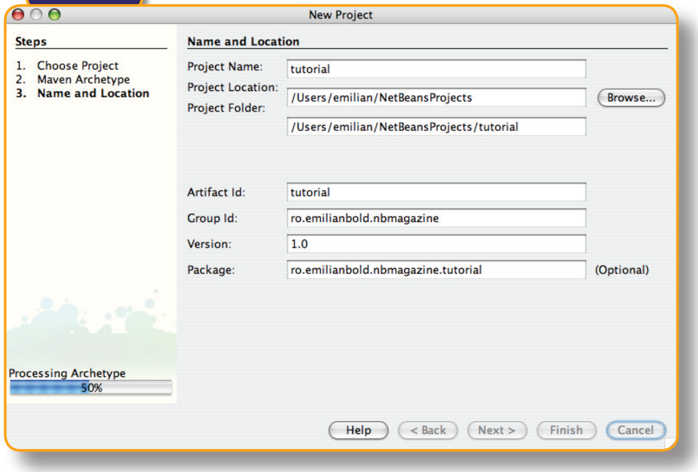


Figure 3. Artifact id, Group id and Version definition

New Project

Steps
1. Choose Project
2. Maven Archetype
3. **Name and Location**

Name and Location

Project Name:	tutorial
Project Location:	/Users/emilian/NetBeansProjects
Project Folder:	/Users/emilian/NetBeansProjects/tutorial
Artifact Id:	tutorial
Group Id:	ro.emilianbold.nbmagazine
Version:	1.0
Package:	ro.emilianbold.nbmagazine.tutorial (Optional)

Processing Archetype 50%

here. The *Artifact Id* is the name for this particular module. The *Version* is used for example for configuration management.

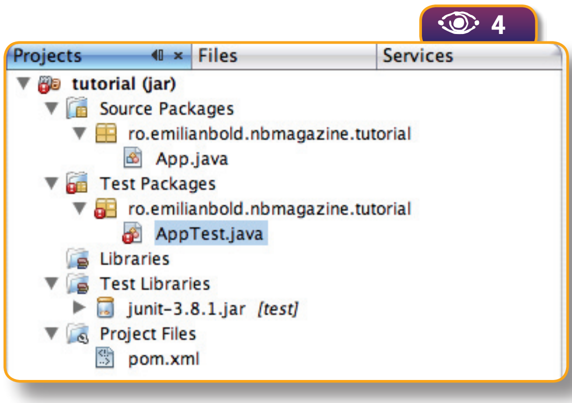
As a result, we have a new project in NetBeans, shown as "tutorial (jar)". You will also notice a package under the *Source Packages* node and another under *Test Packages* (see **Figure 4**). Additionally, you'll have a simple example class and a test, JUnit as a test library (a dependency), and the *pom.xml* file under *Project Files*.

The first strange thing you'll notice if you've never used Maven before is that the project seems to have some errors. The reason in this case is that Maven doesn't come by default with JUnit. JUnit is treated like any other dependency and will need to be downloaded from a repository. Maven takes care of this and any all other dependencies the first time you build the project. It will download the needed artifacts and cache them locally (the default repository being *repo1.maven.org*).

The POM

Let's now open the *Project Files/pom.xml* file, through which you can control all aspects of the project. Changes in the POM will be reflected in the project in the IDE. For example, by changing the **<name/>** element and saving the file, you'll notice that the name of the project changes.

Next we need to change the **<packaging/>** element (whose value is shown in parentheses to the right of the project name). That's because, of course, a NetBeans module isn't distributed as a simple JAR file but as a NBM. So change the packaging to *nbm* and try to build the project. You'll see that it fails miserably.



```
tutorial (jar)
  Source Packages
    ro.emilianbold.nbmagazine.tutorial
      App.java
  Test Packages
    ro.emilianbold.nbmagazine.tutorial
      AppTest.java
  Libraries
  Test Libraries
    junit-3.8.1.jar [test]
  Project Files
    pom.xml
```

Figure 4. New Maven project, with errors that will be solved with the first build

The reason is that no default Maven plugin knows how to handle the *nbm* packaging. We need to add the **nbm-maven-plugin** (which I'll call "NBM Plugin" from now on) inside the **<build/>** element in the POM. See **Listing 1**.

Now the project will build successfully. After the build, switch to the *Files* tab and you'll notice in the *target* folder all the extra artifacts, including the generated NBM file (see **Figure 5**). At this point we have a working module project; by clicking *Run* you'll get a new IDE running, which should include our module among many others.

You might get errors related to Windows paths while trying to run the project. Make sure you don't have spaces in these paths, as these are usually the culprits.

Adding an Action

We will now create a new Platform Action using the *New Action* wizard. The purpose of this Action will be just to inspect that a given service exists and show a dialog. The wizard automatically generates the *Bundle.properties* file in the proper Maven-friendly folder, as well as the Action class. It also changes the layer file and adds the corresponding dependency to the POM.

At this point, any build using Platform APIs will fail, as the Maven project doesn't have a dependency on the needed Platform-specific artifacts. First we need to declare the repository where the NetBeans artifacts are located; see **Listing 2**. Next we include a dependency on **org-openide-util**, which is the module providing the Platform's Actions API. See **Listing 3**. This is equivalent to a dependency added to a normal Platform module. The NBM Plugin will detect that this artifact is a module and configure the proper dependency in the generated build artifact. As before, the project won't initially compile without the dependency; this will be resolved when the files are downloaded on the first build.

Regarding **Listing 3**, if the version RELEASE60 doesn't work for you, try RELEASE60-BETA2 as the new bits might not yet have reached the Maven repository when you read this.

Listing 1. Build configuration for the NBM Plugin

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>nbm-maven-plugin</artifactId>
        <version>RELEASE</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
  ...
 </project>
```

Listing 2. Declaring a default repository for NetBeans artifacts

```
<project>
...
  <repositories>
    <repository>
      <id>netbeans</id>
        <name>Repository for hosting NetBeans API artifacts</name>
        <url>http://deadlock.netbeans.org/maven2/</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
  </repositories>
</project>
```

Now right click on the project node and create an Action with default options. Upon building the project, you might get a warning about Java sources being 1.4 due to some **@Override** annotations which are 1.5 specific. You can just delete the annotations.

All the basic Platform modules have the *groupId* **org.netbeans.api** and the JAR name as the *artifactId*. If you don't know the *groupId/artifactId* for a module, you can try finding it in the repository at *http://deadlock.netbeans.org/maven2*.

Other NetBeans Platform specific settings

So far our module does little to interact with the NetBeans Platform. Sure, by adding a dependency on **org-openide-util** we can already use the lookup service for example, but we can't yet declare a service in the global lookup.

The standard NetBeans way in this case is to place a text file under *META-INF/services*. Luckily this is almost the same under Maven, with a twist: while the Java source files sit under *src/main/java*, all resources must be under *src/main/resources*.

Bleeding edge NetBeans API docs
bits.netbeans.org/dev/javadoc

This Maven-specific separation of resource files from Java source files means extra work if you plan to migrate an existing project to Maven. You'll have to write a script that splits the files that were together in the older project (or do it by hand).

The resources folder may be created outside the IDE, or inside it from the *Files* tab (under *src/main*). It will also be automatically created by the Actions wizard. After this, you should see another node in the *Projects* window called *Other Sources*, containing your resource files (see **Figure 6**).

The *resources* folder only holds resources that belong to the artifact. It does not contain for example the POM file or other Maven configuration files.

The contents and name of the file under *META-INF/services* are the same as usual. Respectively: the service base class or interface; and the fully qualified name of the implementing classes, each in its own line.

The layer file

Now, in order to have menu items or toolbars we need a layer file. The necessary configuration task is letting the build plugin know which is your layer file. In order to do this, you need to create a plugin configuration file (an NBM descriptor), which defines the module metadata you'd expect: cluster name, module type, update center URL, codebase, manifest, etc.

First, create the *src/main/nbm* folder. This is where you'll put the descriptor as a special configuration file (and not in the resources folder). In the new folder, create a file called *module.xml* with contents similar to **Listing 4**.

Listing 3. Adding openide-util as a dependency

```
<project>
...
<dependencies>
...
  <dependency>
     <groupId>org.netbeans.api</groupId>
     <artifactId>org-openide-util</artifactId>
     <version>RELEASE60</version>
   </dependency>
  </dependencies>
...
</project>
```

Listing 4. NBM Plugin descriptor file: module.xml

```
<nbm>
   <moduleType>normal</moduleType>
   <codeNameBase>ro.emilianbold.nbmagazine.tutorial/1</codeNameBase>
   <cluster>nbmagazine</cluster>
   <manifest>src/main/nbm/manifest.mf</manifest>
   <distributionUrl>http://emilianbold.ro/nbmagazine/</distributionUrl>
   <licenseName>GNU GPL 3</licenseName>
   <licenseFile>src/main/nbm/license.txt</licenseFile>
</nbm>
```

The NBM descriptor is capable of holding a lot more data. Please see the NBM Plugin documentation for the full schema.

Next we have to edit the manifest file and declare the layer in the *OpenIDE-Module-Layer* section. While the NBM Plugin lets you declare some module metadata, it currently supports only the manifest file but not the layer. Thus, the *src/main/nbm/MANIFEST.MF* file defined in *module.xml* should be created with this line content (in a single line):

```
OpenIDE-Module-Layer:
  ro/emilianbold/nbmagazine/tutorial/layer.xml
```

We know that anything that isn't a Java source class must be placed in the resources folder; the layer file is no exception as it will also be part of the final build artifact. Now it's time to rebuild and re-run the project. You'll be happy to notice that the layer is properly registered, that our Action is working, and also that we can declare services in *META-INF/services*.

With the configuration done so far, the manifest, layer and NBM descriptor files, plus some dependencies, we have covered about 90% of the Platform development cases. Next we'll talk about JavaHelp modules, branding and suites, which should bring us to 100%.

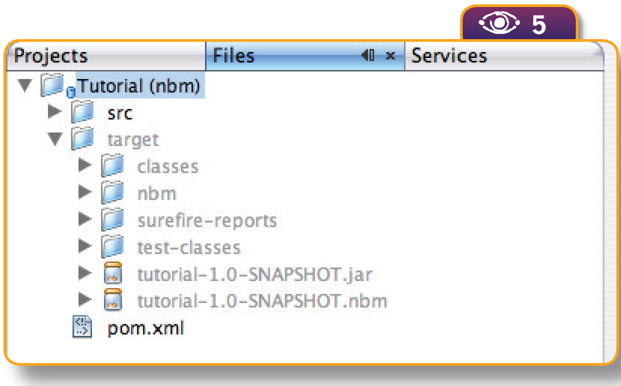
Help modules

The NetBeans Platform has excellent JavaHelp support via NetBean's standard build harness; the NBM Plugin also supports building modules with JavaHelp documentation.

First, you'll need a new empty Maven project configured like the previous one (but without the Action), containing a NBM descriptor and an empty layer file. I'll assume "ro.emilianbold.nbmagazine.tutorial" as *groupId* and "help" as *artifactId*. Also, the layer must declare the reference to the JavaHelp docs (see **Listing 5**).

The *helpset.xml* file (see **Listing 6**) just contains a reference to the location of the helpset configuration. The reason for doing this is that the documentation won't actually be in the main JAR artifact but in a separate JAR (the kind of JAR you see in the *docs* folder in the cluster).

Now we get to the actual JavaHelp files. First we need to create a new folder: *src/main/javahelp/${groupId}/${artifactId}/docs* (with our *groupId/artifactId*, that would be *src/main/javahelp/ro/emil-*


5

Projects | Files | Services

```
▼ Tutorial (nbm)
  ▶ src
  ▼ target
    ▶ classes
    ▶ nbm
    ▶ surefire-reports
    ▶ test-classes
    ▶ tutorial-1.0-SNAPSHOT.jar
    ▶ tutorial-1.0-SNAPSHOT.nbm
    pom.xml
```

Figure 5. NBM artifact in target folder

Listing 5. layer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE filesystem PUBLIC "-
  //NetBeans//DTD Filesystem 1.1//EN"
    "http://www.netbeans.org/dtds/filesystem-1_1.dtd">
<filesystem>
  <folder name="Services">
    <folder name="JavaHelp">
      <file name="helpset.xml" url="helpset.xml">
        <attr name="position" stringvalue="1000"/>
      </file>
    </folder>
  </folder>
</filesystem>
```

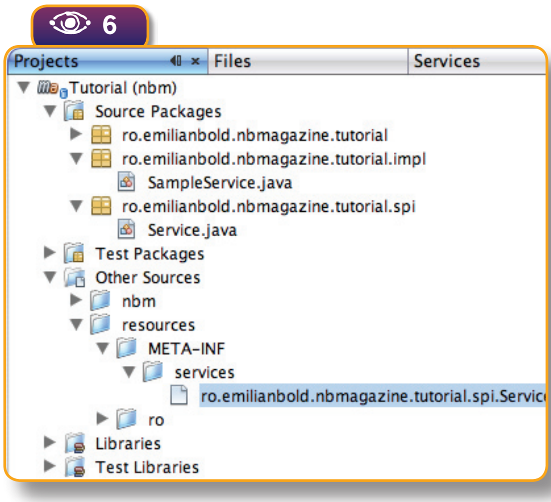
Listing 6. helpset.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE helpsetref PUBLIC "-
  //NetBeans//DTD JavaHelp Help Set Reference 1.0
  //EN" "http://www.netbeans.org/dtds/helpsetref-1_0.dtd">
<helpsetref
    url="nbdocs:/ro/emilianbold/nbmagazine/tutorial/help/docs/hs.xml"/>
```

mevenide.codehaus.org/m2-site/index.html

Mevenide NetBeans integration modules

Figure 6. Other Sources node


6

Projects | Files | Services

- Tutorial (nbm)
  - Source Packages
    - ro.emilianbold.nbmagazine.tutorial
    - ro.emilianbold.nbmagazine.tutorial.impl
      - SampleService.java
    - ro.emilianbold.nbmagazine.tutorial.spi
      - Service.java
  - Test Packages
  - Other Sources
    - nbm
    - resources
      - META-INF
        - services
          - ro.emilianbold.nbmagazine.tutorial.spi.Servic
      - ro
  - Libraries
  - Test Libraries

Listing 7. JavaHelp files

Map (map.xml)

```
<!-- ... XML/DOCTYPE header -->
<map version="2.0">
  <mapID target="about" url="about.html"/>
</map>
```

Table of contents (toc.xml)

```
...
<toc version="2.0">
  <tocitem text="Maven2 in NetBeans ?">
    <tocitem text="About" target="about"/>
  </tocitem>
</toc>
```

Index (idx.xml)

```
...
<index version="2.0">
  <indexitem text="About Maven2 Javahelp" target="about"/>
</index>
```

Helpset (hs.xml)

```
...
<helpset version="2.0">
  <title>Help</title>
  <maps>
    <homeID>about</homeID>
       <mapref location="map.xml"/>
  </maps>
  <view mergetype="javax.help.AppendMerge">
    <name>TOC</name>
      <label>Table of Contents</label>
      <type>javax.help.TOCView</type>
      <data>toc.xml</data>
  </view>
  <view mergetype="javax.help.AppendMerge">
    <name>Index</name>
    <label>Index</label>
    <type>javax.help.IndexView</type>
    <data>idx.xml</data>
  </view>
  <view>
    <name>Search</name>
    <label>Search</label>
    <type>javax.help.SearchView</type>
    <data
       engine="com.sun.java.help.search.DefaultSearchEngine">
       JavaHelpSearch
    </data>
  </view>
</helpset>
```

ianbold/nbmagazine/tutorial/help/docs.) Then create the various JavaHelp files (see **Listing 7**). Compile and run the project, and you'll see that the help works (see **Figure 7**).

OK, remember you shouldn't copy-and-paste? This is exactly what I did to bootstrap this module and get the JavaHelp files. After you obtain the base files, you just need to add the new HTML files and entries to the map.

Library wrappers

So far we've seen how to declare a normal module and add dependencies. However, a module may also "wrap" an existing JAR and export part or all of its packages. Let's see how to do this.

Adding a dependency to a third-party JAR can be done the normal Maven 2 way (see **Listing 8**). You just need to remember to have a repository declared in the POM if the JAR is not in the standard repository.

The NBM Plugin will automatically add the JAR to the NBM, but there will be no public packages so far, so it can only be used internally. Sadly, the public packages will have to be manually added to the manifest (see **Listing 9**), which is quite painful but should be a one-time job.

Remember that the manifest file is quite finicky with line lengths, so you might need to break it into multiple lines (each one starting with a single space).

Normally leaving an empty *OpenIDE-Module-Public-Packages* means that *all* packages will be public. Note that though this is good for normal modules, it won't work for library wrappers.

The module suite

We've already seen how to create individual modules, module wrappers and documentation modules, but we still need to put them somehow in a suite. The solution is to rely on Maven again and use an *aggregating project*. This must have the POM packaging and list each of the contained sub-modules (see **Listing 10**).

While the NBM Plugin is able to generate the whole suite cluster with the **cluster** goal, you still have to configure it to run during the build project (see **Listing 11**). Note that the **<module/>** elements point to the actual disk folders, as opposed to the normal way of using *groupId:artifactId:version* for dependencies.

In the configuration file in **Listing 11,** I first register the NBM Plugin as a build plugin extension. Then I define the enabled clusters, as well as the *brandingToken* (needed for branding) and *keystorealias*. All this information is used by the **cluster** goal, which is responsible for generating the Platform-compatible cluster. Next, with the **<execution/>** element, I register the plugin to run during the build and execute the **cluster** goal.

This way, the plugin will run each time I build the aggregating project and generate the proper cluster. You can run the application now and notice that it's quite simple (it only uses the Platform cluster), has the help working, and even our little Action which uses the Lookup service works (see **Figure 8**).

Listing 8. A non-NBM (plain JAR) dependency

```
<project ...>
  ...
  <dependencies>
    <dependency>
       <groupId>net.java.dev.swing-layout</groupId>
          <artifactId>swing-layout</artifactId>
          <version>1.0</version>
    </dependency>
  </dependencies>
  ...
</project>
```

Listing 9. MANIFEST.MF for holding the public packages

```
Manifest-Version: 1.0
...
OpenIDE-Module-Public-Packages: org.jdesktop.layout.*
...
```

Listing 10. Aggregating project

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>ro.emilianbold.nbmagazine</groupId>
  <artifactId>suite</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>suite</name>
  <modules>
    <module>../help</module>
    <module>../tutorial</module>
  </modules>
...
</project>
```

Listing 11. NBM Plugin configuration

```
<project ...>
...
  <build>
     <plugins>
       <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>nbm-maven-plugin</artifactId>
         <version>LATEST</version>
         <extensions>true</extensions>
         <inherited>false</inherited>
         <configuration>
            <keystorealias>nbmagazine</keystorealias>
            <brandingToken>nbmagazine</brandingToken>
            <enabledClusters>
              <enabledCluster>platform7
              </enabledCluster>
              <enabledCluster>nbmagazine
              </enabledCluster>
            </enabledClusters>
          </configuration>
          <executions>
            <execution>
              <id>cluster</id>
              <phase>process-resources</phase>
              <goals>
                <goal>cluster</goal>
              </goals>
            </execution>
          </executions>
       </plugin>
     </plugins>
  </build>
</project>
```

www.emilianbold.ro

The author's homepage

Branding

The last piece of the puzzle is branding. Support via actual wizards is totally missing to this date, so it's back to manual work or copy-pasting from another project. You'll need a *src/main/nbm-branding* folder where all the branding sources will reside. The folder's structure should be the same as the one used by the Ant-based build harness. Also, the POM must be changed to configure the **nbm:branding** goal as in **Listing 12**. The end result is a branded application as seen in **Figure 9**.

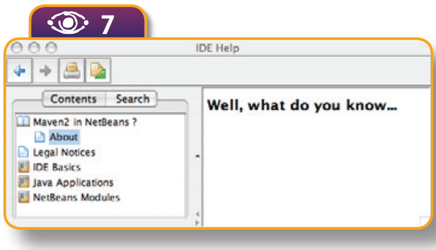


Figure 7. Our help content registered in the main Platform help

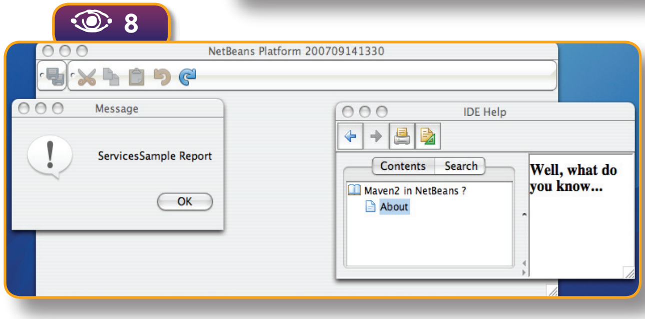


Figure 8. Module suite with Help

Conclusions

Using Maven to build NetBeans Platform applications is no longer an obscure task. The current integration makes Maven-based projects almost on par with standard IDE projects and the gap is narrowing. So, if you like Maven but couldn't use it before with NetBeans IDE, or you do NetBeans Platform development but can't use the IDE for some reason, rest assured that there's a good and rapidly improving solution now.

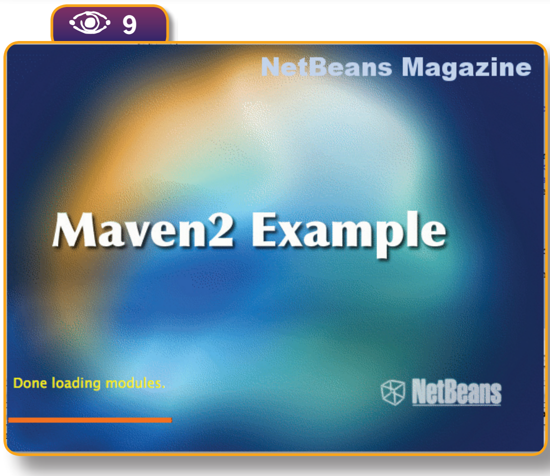


Figure 9. The branded splash screen

Emilian Bold (*emilian.bold.public@gmail.com*) is a Java and NetBeans Platform consultant from Timisoara, Romania, as well as member of the NetBeans Dream Team. He has been working with the NetBeans Platform since version 3.6, starting with a project at Alcatel Romania (now Alcatel-Lucent) and owns a NetBeans Platform-focused consulting company.

Listing 12. Running the branding goal in the process-resources phase

```
<project ...>
  <build>
    <plugins>
       <plugin>
         <groupId>org.codehaus.mojo</groupId>
            <artifactId>nbm-maven-plugin</artifactId>
            <version>RELEASE</version>
            <extensions>true</extensions>
          <configuration>
            <brandingSources>${basedir}/src/main/nbm-branding</brandingSources>
            <brandingToken>nbmagazine</brandingToken>
            <cluster>nbmagazine</cluster>
            <nbmBuildDir>${project.build.directory}/nbm</nbmBuildDir>
          </configuration>
          <executions>
            <execution>
            <id>branding</id>
            <phase>process-resources</phase>
            <goals>
            <goal>branding</goal>
            </goals>
            </execution>
          </executions>
        </plugin>
       ...
       /plugins>
  </build>
  ...
</project>
```

NetBeans magazine

In-depth, no-fluff articles for the full gamut of NetBeans developers

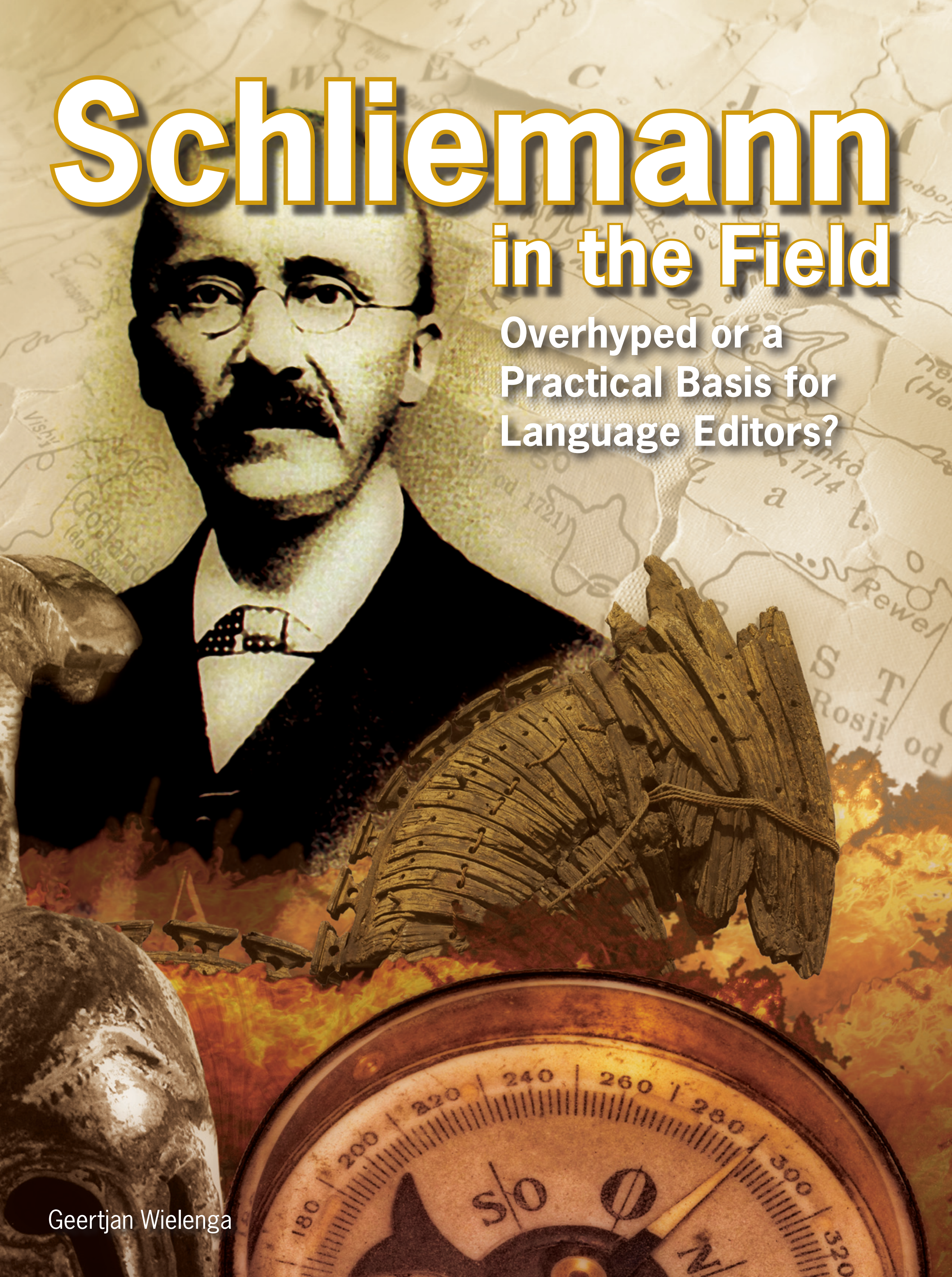
- Platform APIs
- Java SE development
- Profiling
- Mobility
- Dynamic languages
- Modules
- Visual Web
- Web Services
- Matisse
- C/C++
- SOA
- Java EE development
- Ruby
- UML
- Community

NetBeans magazine
December . 2007
The Best Ruby IDE
Dynamic web development and the IDE's Ruby features
Fluent in NetBeans
How global localization efforts are expanding the IDE's reach
Advanced Profiling
Real-world explorations with the NetBeans Profiler
Schliemann in the Field
People and projects working to multiply language support
Expand your development Horizons with NetBeans 6.0
Creating RESTful Web Services
Module Development with Maven
Get to Know GroupLayout
Plug-in Module Quick Start
Bringing Light to Java Development
Java EE 5 in Practice
Not Just an IDE
UML Modeling in Action
Boosting Enterprise Productivity
May . 2007
Core NetBeans 6.0 Features
Introducing C/C++ Pack
The blueMarine Project
OpenOffice.org Integration
Project Schliemann
Mobility Pack in Practice
New UI Design Features
Visual Web Development
Reach Out with the IDE and Platform

Check out previous issues at
netbeans.org/community/magazine

Schliemann in the Field

Overhyped or a Practical Basis for Language Editors?



Geertjan Wielenga

In the previous issue of the NetBeans Magazine, we highlighted a new and simplified approach to providing syntax highlighting, code completion, and other language-support features for programming languages. Instead of a multi-file Java-based approach using complex NetBeans APIs, we have a "new way" known as *Schliemann*. The idea behind it is to have a simple set of declarations in a single file, written in the form of regular expressions that describe the language tokens and grammar, together with their assignment to features in the IDE. Using these declarations, the IDE is then able to do the lexing and parsing for you, so there's no need to provide such functionality yourself.

But how viable is this approach? Can it be used in real-life scenarios? Or is it just a fuzzy story without real usage in the workplace? In this article, we investigate a number of actual Schliemann-based implementations by NetBeans developers around the world.

At the time these implementations were started, the day was still young for the Schliemann framework, which was created for NetBeans 6.0. Despite that, several language developers who liked living on the edge had already started working with milestones and daily builds, writing editors for their favorite languages even before the framework on which they were basing their project was actually complete.

There are dangers in this, of course, but Schliemann has been stable enough to allow it, judging by the advanced results some of these implementations already provide. Each implementation is briefly discussed below, in the context of the developer who implemented it, and accompanied by some screenshots to give you an impression of the achievements thus far.

Erlang Language Support, by Caoyuan Deng

Erlang is a Functional Programming language with many dynamic features, such as being dynamically typed, with hot code swapping. "Erlang is ideal for soft real-time, heavy concurrent-load network applications," says Caoyuan Deng, who is adding Erlang support to NetBeans. In **Figure 1** you can see a segment of the editor that Caoyuan has created, showing syntax coloring and code folding.

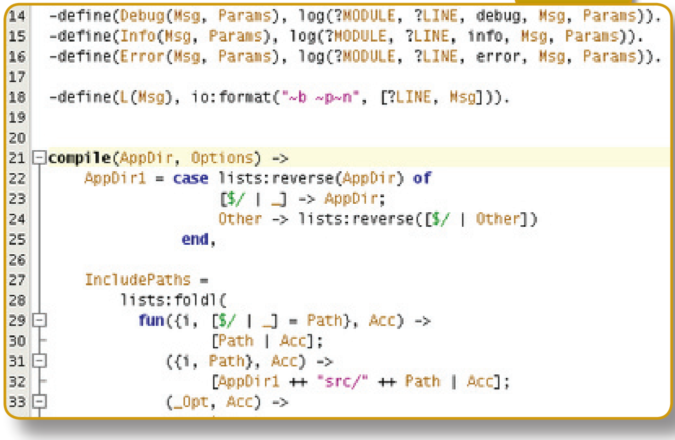
When asked about the usability of the framework, Caoyuan says that, "with Schliemann you define a grammar file for a new language and you not only get features like syntax highlighting, indentation and code-folding, but also a visual incremental AST tree. This is helpful for writing IDE support for new languages. When you're typing code, Schliemann parses the syntax incrementally in real-time, and you know immediately a lot of context information about the code, such as whether there is a variable in the current function's scope."

OpenGL Shading Language Support, by Michael Bien

OpenGL Shading Language (GLSL) is a language for graphical effects, such as lighting, fog, water, reflections, and refraction. It defines how an element is rendered in an OpenGL scene. Michael Bien created an editor for this language as part of his recently released NetBeans OpenGL Pack (see it at work in **Figure 2**).

"I am currently working on a Schliemann-based language valida-

1


```
-define(Debug(Msg, Params), log(?MODULE, ?LINE, debug, Msg, Params)).
-define(Info(Msg, Params), log(?MODULE, ?LINE, info, Msg, Params)).
-define(Error(Msg, Params), log(?MODULE, ?LINE, error, Msg, Params)).

-define(L(Msg), io:format("~b ~p~n", [?LINE, Msg])).


compile(AppDir, Options) ->
    AppDir1 = case lists:reverse(AppDir) of
                  [$/ | _] -> AppDir;
                  Other -> lists:reverse([$/ | Other])
              end,

    IncludePaths =
        lists:foldl(
          fun({i, [$/ | _] = Path}, Acc) ->
                  [Path | Acc];
             ({i, Path}, Acc) ->
                  [AppDir1 ++ "src/" ++ Path | Acc];
             (_Opt, Acc) ->
```

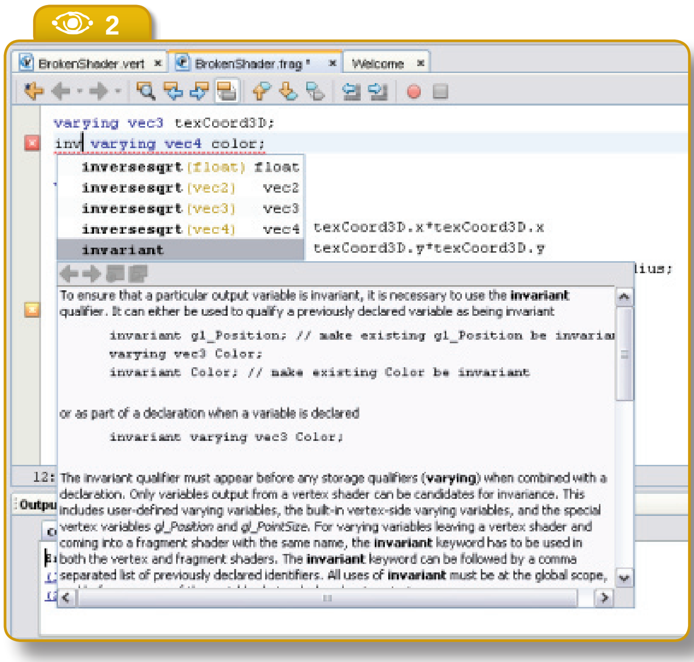
wiki.netbeans.org/wiki/view/Schliemann

Schliemann Wiki

languages.netbeans.org

Generic Languages Framework (Project Schliemann)

Figure 1. Editing Erlang in NetBeans



```
varying vec3 texCoord3D;
inv varying vec4 color;
    inversesqrt (float) float
    inversesqrt (vec2)    vec2
    inversesqrt (vec3)    vec3
    inversesqrt (vec4)    vec4  texCoord3D.x*texCoord3D.x
    invariant                   texCoord3D.y*texCoord3D.y
                                                           lius;
```

To ensure that a particular output variable is invariant, it is necessary to use the **invariant** qualifier. It can either be used to qualify a previously declared variable as being invariant

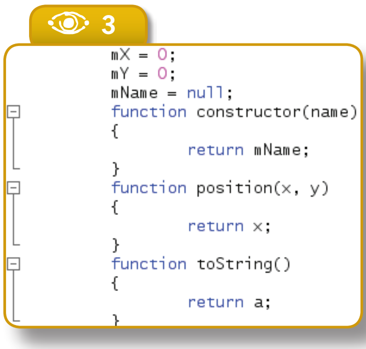
```
invariant gl_Position; // make existing gl_Position be invaria
varying vec3 Color;
invariant Color; // make existing Color be invariant
```

or as part of a declaration when a variable is declared

```
invariant varying vec3 Color;
```

The invariant qualifier must appear before any storage qualifiers (**varying**) when combined with a declaration. Only variables output from a vertex shader can be candidates for invariance. This includes user-defined varying variables, the built-in vertex-side varying variables, and the special vertex variables *gl_Position* and *gl_PointSize*. For varying variables leaving a vertex shader and coming into a fragment shader with the same name, the **invariant** keyword has to be used in both the vertex and fragment shaders. The **invariant** keyword can be followed by a comma separated list of previously declared identifiers. All uses of **invariant** must be at the global scope,

Figure 2. GLSL support



```
        mX = 0;
        mY = 0;
        mName = null;
        function constructor(name)
        {
                return mName;
        }
        function position(x, y)
        {
                return x;
        }
        function toString()
        {
                return a;
        }
```

Figure 3. MiniD programming in action

tor for GLSL in combination with the NetBeans Lexer API, for an alternative to validation via compilation," says Michael. "This is an important feature because GLSL shader programs are compiled by the VGA driver, and this can lead to different results on different hardware (or even driver versions)."

About his experience with Schliemann, he says that the declarative language the framework supports is very intuitive and provides access to most of the features you would expect from an IDE's editor, without limiting the developer. He also highlights that you can call Java code from inside the Schliemann definition file.

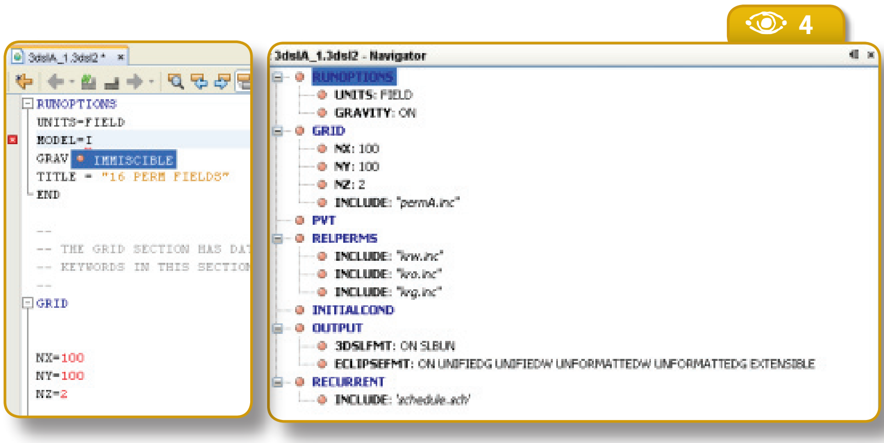
D and MiniD Language Support, by Björn Lietz-Spendig

Giving his reasons for creating editor support for D and the MiniD scripting language, Björn says: "I don't like C++, especially its preprocessor and template language. But, despite being a database developer who is used to working with RAD tools, I have a need for low-level programming. And now we have a stable D 1.x, including a module system, single inheritance, interfaces, garbage collector, design by contract, class invariants, in addition to templates as a natural part of the language." **Figure 3** shows the results so far for a simple MiniD code fragment.

Commenting on the future of his editor support, Björn says: "The D implementation could open new horizons to NetBeans developers. Can you imagine being able to use a C++ class directly from Java and vice versa? With the D language plugin you can call D classes (via TioLink) and D can call Java classes. You have the power to create platform-independent eye-candy applications based on the NetBeans Platform, while having seamless integration with D".

3DSL Support, by Max Calderoni & Marco Thiele

"We are currently developing a plugin supporting an internal format for a file that in our industry (oil) is currently edited by hand," says Max Calderoni. "It's



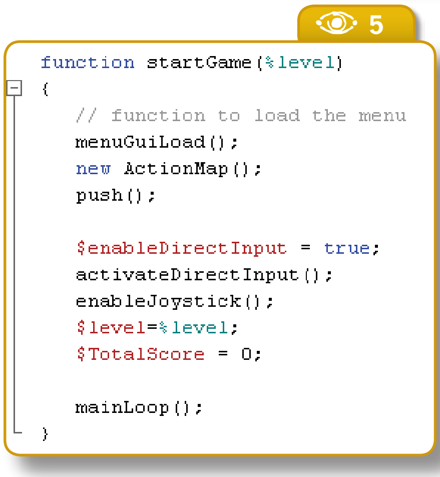
```
3dslA_1.3dsl2 *
RUNOPTIONS
UNITS=FIELD
MODEL=I
GRAV  IMMISCIBLE
TITLE = "16 PERM FIELDS"
END

--
-- THE GRID SECTION HAS DAT
-- KEYWORDS IN THIS SECTION
--
GRID


NX=100
NY=100
NZ=2
```

```
3dslA_1.3dsl2 - Navigator
RUNOPTIONS
    UNITS: FIELD
    GRAVITY: ON
GRID
    NX: 100
    NY: 100
    NZ: 2
    INCLUDE: "permA.inc"
PVT
RELPERMS
    INCLUDE: "krw.inc"
    INCLUDE: "kro.inc"
    INCLUDE: "krg.inc"
INITIALCOND
OUTPUT
    3DSLFMT: ON SLBLN
    ECLIPSEFMT: ON UNIFIEDG UNIFIEDW UNFORMATTEDW UNFORMATTEDG EXTENSIBLE
RECURRENT
    INCLUDE: 'schedule.sch'
```

Figure 4. Support for 3DSL in complex, structured input files

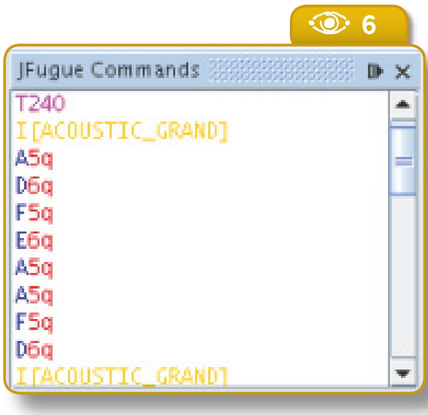


```
function startGame(%level)
{
   // function to load the menu
   menuGuiLoad();
   new ActionMap();
   push();

   $enableDirectInput = true;
   activateDirectInput();
   enableJoystick();
   $level=%level;
   $TotalScore = 0;

   mainLoop();
}
```

Figure 5. TorqueScipt editing, with syntax coloring and code folding support



```
JFugue Commands
T240
I[ACOUSTIC_GRAND]
A5q
D6q
F5q
E6q
A5q
A5q
F5q
D6q
I[ACOUSTIC_GRAND]
```

Figure 6. Syntax-colored JFugue notation

the input file for a numeric simulator, not a programming language, and is called 3DSL. The simulator is used for oil reservoir analysis". Such input files, the authors continue, are difficult to edit even for advanced users, "so it would be a blessing to have code folding, code completion, syntax highlighting and inline documentation for 3DSL files". And they are well on that track using Schliemann (see **Figure 4**).

TorqueScript Support, by Mohamed el Shami

TorqueScript is the scripting language of the Torque Game Engine and the Torque Game Builder (TGB), by GarageGames. TorqueScript has a C++-like syntax and includes features like Datablocks, Objects, and Namespaces. "What I've done so far with TorqueScript is only a start," says Mohamed. "Since I've been learning along the way, I'm firstly working to complete the grammar rules. Then I'd like to add support to run ning the game from within the editor." **Figure 5** shows an example with TorqueScript.

"I'd also like to add a project template for a new TGB game project which creates the game project structure," says Mohamed. "Later, the plan is to include advanced features like debugging, refactoring, and a dynamically populated auto-complete list from the game engine APIs."

JFugue Support, by the author

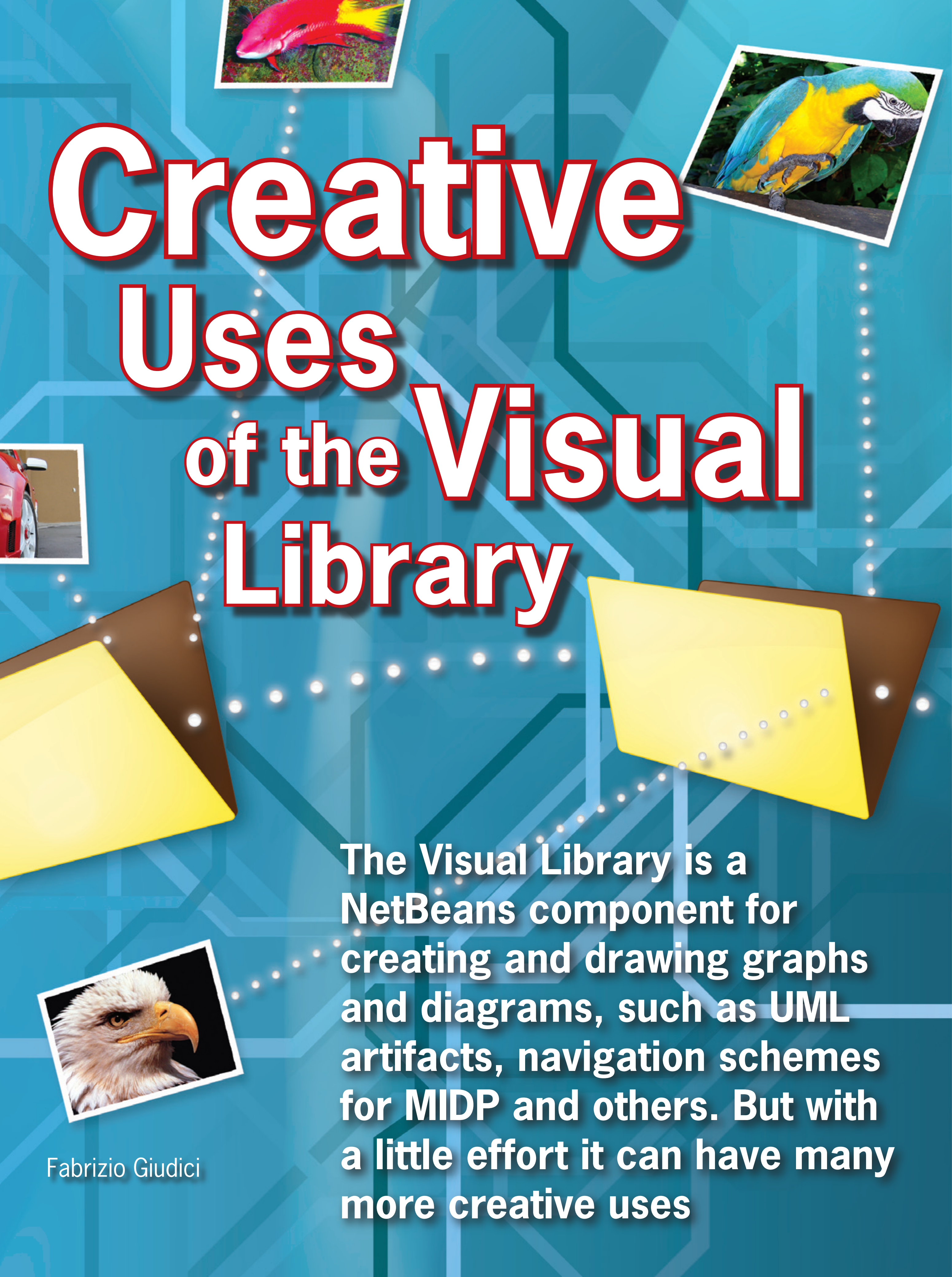
JFugue notation provides the music strings that let you specify notes, chords and instruments when you use the JFugue API for music programming. The notation is simple, but syntax coloring is always useful. In the example shown in **Figure 6**, the formatting is applied to a **JEditorPane**. This is simple to do, by assigning the MIME type as the value of the **ContentType** property. As a result, the notes and their duration can be distinguished from each other, and from the instrument selection (shown in yellow).

Conclusions

Despite its youth, the Schliemann framework is proving extremely useful in a number of varying scenarios. It is easy to use and the results are surprisingly effective. Why not take it for a spin?

Geertjan Wielenga (*geertjan.wielenga@sun.com*) is a technical writer for the NetBeans project. He mainly focuses on the NetBeans Platform and is a co-author of the book "Rich-Client Programming: Plugging into the NetBeans Platform". He lives in Prague, in the Czech Republic.

Creative Uses of the Visual Library



The Visual Library is a NetBeans component for creating and drawing graphs and diagrams, such as UML artifacts, navigation schemes for MIDP and others. But with a little effort it can have many more creative uses

Fabrizio Giudici

The Visual Library has been conceived mainly for building, handling and rendering graphs, including flow and UML diagrams (it has been originally developed for the NetBeans Mobility Pack, but from NetBeans 6.0 on it's part of the NetBeans Platform APIs). I like to think of the Visual Library more generally, however, as an API for creating interactive "whiteboards" where you can place, move, and rearrange items visually.

From this perspective, the Visual Library reveals all its power, as modern UIs are each day more focused on the concept of modeling "real-life" objects that can be moved around. Thus it found its way into blueMarine, an open-source photo management application I created which is based on the NetBeans Platform (see my article in Issue 3 of NetBeans Magazine for more about this tool).

In blueMarine, the Visual Library is used to implement a virtual version of a photographer's "Light Table" – a place where photos can be laid out and rearranged (see **Figure 1**). It's also the basis for an advanced geotagging component (**Figure 2**). In this article I'll describe the Light Table, which is the simpler of the two Visual Library-based components, but complex enough that we can show in practice many features of the API.

Though all the examples illustrated in this article refer to a NetBeans Platform application, you can use the Visual Library in plain Swing apps by adding a couple of JAR files to the classpath.

2

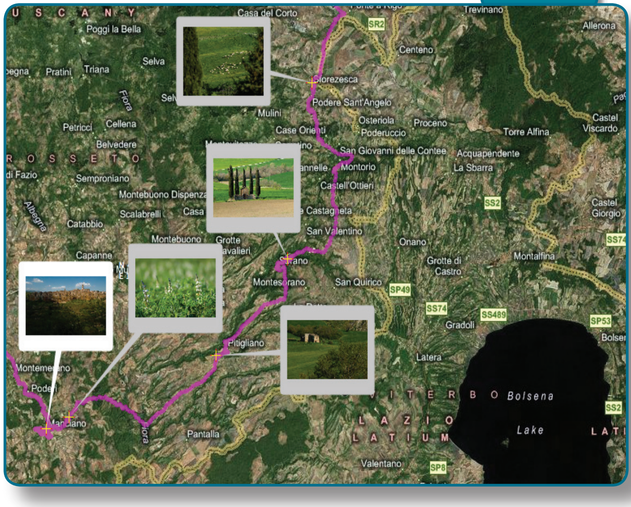


Figure 2. The Geotagging Component

graph.netbeans.org Homepage for the Visual Library

1

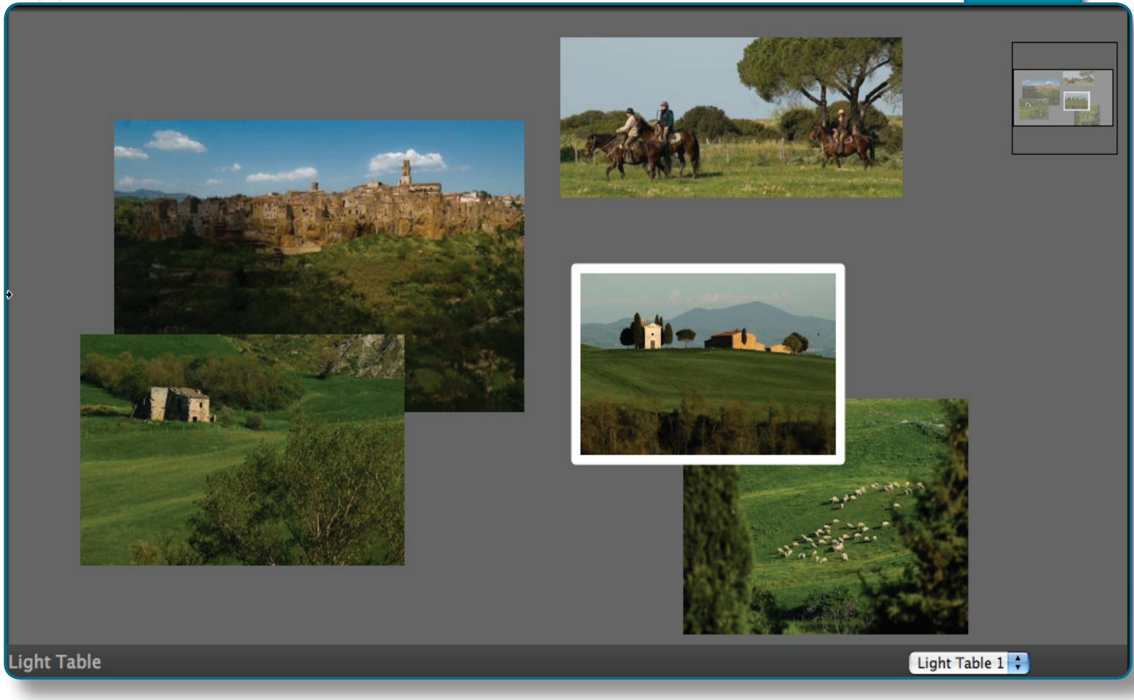


Figure 1. The Light Table

Basic concepts

Let's first introduce some key concepts of the Visual Library.

Widgets (the **org.netbeans.api.visual.widget.Widget** class and its descendants) represent a diagram's nodes. A widget can consist of a simple drawing, a piece of text, an icon, or a group of these basic elements. It can also wrap a Swing component. You may need to subclass **Widget** for special purposes, but in most cases you'll be fine with one of the provided widget classes such as **ImageWidget** and **LabelWidget**.

Connection widgets, in particular, represent arcs that connect pairs of other widgets. They are usually drawn as arrows (with some graphic variants for line caps). Also, their paths can be "routed" using different algorithms, for example to avoid clutter in diagrams.

Other essential elements are **Actions** and **ActionFactories.** You probably won't be satisfied by just creating a diagram and staring at it, so there's plenty of support for making diagrams dynamic and interactive. It's possible to create, delete and select widgets, and have them change appearance when you hover over them. You can additionally drag, connect and disconnect widgets. The Visual Library provides **Action**s and corresponding factories to perform these tasks, and allows you to customize their behavior.

Finally, a **scene** (**org.netbeans.api.visual.widget.Scene**) is the container for everything (in my initial analogy it represents the actual "whiteboard").

In this article I assume the reader is quite confident with these basic concepts. There are already various tutorials available on the library's website where you can learn the essentials. Here I present a more advanced, let's say "creative", use of the library.

Getting the code

You can download the full working code described in this article by using Subversion:

```
svn checkout \ https://bluemarine-incubator.dev.java.net/svn/
    bluemarine-incubator/trunk/src/LightTable -r 232 \
    --username guest
```

Revision 232 is the one matching the code listings in this article.

The Light Table

The Light Table lives inside a **TopComponent** (in a plain Swing application you would use a **JFrame** or a **JPanel**).

If you're not familiar with the NetBeans Platform, a **TopComponent** is sort of a hybrid between a **JFrame** and a **JPanel**. It is normally docked, thus behaving as a **JPanel**, but can be undocked and float around like a **JFrame**. It is usually the container used for a user interface in NetBeans Platform applications.

Listing 1. Basic objects for the Visual Library in LightTableTopComponent.java

```
private final ObjectScene scene = new ObjectScene();
private final JComponent view = scene.createView();
private final LayerWidget mainLayer = new LayerWidget(scene);
private final JComponent satelliteView = scene.
createSatelliteView();
```

Listing 2. Initializing components in LightTableTopComponent

```
spScrollPane.setViewportView(view);
pnLayeredPane.add(spScrollPane, JLayeredPane.DEFAULT_LAYER);

// pnSatelliteView is a JPanel wrapping the satelliteView
pnLayeredPane.add(pnSatelliteView, JLayeredPane.PALETTE_LAYER);
scene.addChild(mainLayer);
scene.getActions().addAction(ActionFactory.createZoomAction());
```

A number of required objects are initialized in the **LightTableTopComponent** (see **Listing 1**):

- **scene** – An instance of **ObjectScene**, a class that keeps associations between widgets and the objects that model them. This facility is quite useful for implementing the MVC pattern.
- **view** – A Swing **JComponent** that renders the objects in the **scene**. You obtain it by asking the scene object for its creation.
- **mainLayer** – The Visual Library

usually organizes widgets in different layers. This can be handy especially for improving performance of graphs with connections. Our Light Table does not need connections so we're fine with a single layer.

• **satelliteView** – This **JComponent** will render a "bird's eye view" of the scene, useful if you're going to create a large scene that extends beyond the screen size.

Listing 2 shows how the initialization is completed. You usually need to place the scene in a **JScrollPane**, to allow users to navigate it. We also use a **JLayeredPane** to render the satellite view above the scene in a corner. In the last line we enable zooming by creating a "zoom action" through the **ActionFactory** class and adding it to the scene.

Adding and removing objects

We also need to include a couple of methods for adding and removing photos in the Light Table; see **Listing 3**. In this code, when a new **DataObject** is added we perform the following steps (inside the **internalAdd()** method):

1. Create the widget that represents the **DataObject** in the scene (the **ThumbnailWidget**, discussed below).
2. Assign its initial position (with some code needed to convert a Swing **Point** to a proper value in the Visual Library's scene model coordinates).
3. Add the widget to the layer.
4. Add the widget and the **DataObject** to the scene (which binds them together).
5. Define the dynamic behavior of the widget by adding some actions. Many actions are created through **ActionFactory**, while others can be instantiated with specific methods of the **scene** object.
6. Finally, call **scene.validate()**. After one or more widgets are changed, the scene needs to be revalidated. The Visual Li-

Listing 3. Adding and removing objects from the Light Table

```
public void addDataObject (final DataObject dataObject) {
  internalAdd(dataObject, new Point(100, 100)); // default coordinates
}

public void removeDataObject (final DataObject dataObject) {
  internalRemove(dataObject);
}

private void internalAdd (final DataObject dataObject, final Point viewLocation) {
  final ThumbnailWidget widget = new ThumbnailWidget(scene, dataObject.getNodeDelegate());
  final Point sceneLocation = scene.convertViewToScene(viewLocation);
  final Point localLocation = mainLayer.convertSceneToLocal(sceneLocation);

  widget.setPreferredLocation(localLocation);
  widget.setUnselectedBorder(EMPTY_BORDER);
  mainLayer.addChild(widget);
  scene.addObject(dataObject, widget);

  // resizeStrategy is described in Listing 4
  widget.getActions().addAction(ActionFactory.createResizeAction(resizeStrategy, null));
  widget.getActions().addAction(ActionFactory.createMoveAction());
  widget.getActions().addAction(scene.createSelectAction());
  widget.getActions().addAction(scene.createObjectHoverAction());
  // bringToFrontAction is described in Listing 5
  widget.getActions().addAction(bringToFrontAction);
  scene.validate();
}

private void internalRemove (final DataObject dataObject) {
  final List<Widget> widgets = scene.findWidgets(dataObject);
  scene.removeObject(dataObject);

  //removeObject() does not remove widgets
  for (final Widget widget : widgets) {
    widget.removeFromParent();
  }
}
```

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

The author's blogs

brary usually does this automatically, but as we're writing customized Swing code that performs a change we need to do this manually.

Before we go on, let me discuss some basic concepts about the way the NetBeans Platform implements the MVC pattern. In the Platform, the **DataObject** class is used to model a domain object, and an associated class – **Node** – is used to model its representation inside a view (a list, a tree, etc.). For example, **Node**s contain a text label, an icon, and a list of associated actions which can be activated by a popup menu.

It's a very good thing to have two distinct classes, since you can have multiple **Node**s for each **DataObject**. This lets you create different representations of the same domain object. You usually subclass **DataObject** and add your application-specific logic. For example, blueMarine defines a **PhotoDataObject** class which contains code for reading and writing an image. However, you won't see this class in the code in this article because I'm following a best practice of keeping models as general as possible, by working with plain **DataObject**s and delegating everything to the related **Node** classes. Thus, I could use the same code, e.g. for rendering movies (with a **MovieDataObject**) or other visual documents.

Now let's go back to the **LightTableTopComponent**.

Widget behavior

We want users to be able to select our widget, resize it by dragging its borders, and move it by dragging its contents. Also, the widget should change appearance when the mouse hovers over it, and come to the top of the stack when clicked. For movement, selection and hovering, adding

Wiki for Ruby support in NetBeans

wiki.netbeans.org/wiki/view/Ruby

Listing 4. Providing a custom ResizeStrategy for preserving aspect ratio

```
private final static ResizeStrategy resizeStrategy = new ResizeStrategy() {
  public Rectangle boundsSuggested (final Widget widget,
      final Rectangle originalBounds,
      final Rectangle suggestedBounds,
      final ResizeProvider.ControlPoint controlPoint)
  {
    final Rectangle result = new Rectangle(suggestedBounds);
    final Thumbnail thumbnail = widget.getLookup().lookup(Thumbnail.class);

    // We could compute aspectRatio from originalBounds,
    // but rounding errors would accumulate.
    if (thumbnail != null)  {
      // isImageAvailable() doesn't guarantee the image is online
      final BufferedImage image = thumbnail.getImage();

      if (image != null) {
        final Insets insets = widget.getBorder().getInsets();
        final int mw = insets.left + insets.right;
        final int mh = insets.bottom + insets.top;
        final int contentWidth = result.width - mw;
        final int contentHeight = result.height - mh;
        final float aspectRatio = (float) image.getHeight()/image.getWidth();
        final double deltaW = Math.abs(suggestedBounds.getWidth() - originalBounds.getWidth());
        final double deltaH = Math.abs(suggestedBounds.getHeight() - originalBounds.getHeight());

        if (deltaW >= deltaH) { // moving mostly horizontally
          result.height = mh + Math.round(contentWidth * aspectRatio);
        }
        else { // moving mostly vertically
          result.width = mw + Math.round(contentHeight / aspectRatio);
        }
      }
    }
    return result;
  }
};
```

predefined actions suffice. For resizing support though, the **ActionFactory.createResizeAction()** method won't do: it lets you arbitrarily change the widget's dimensions, but the photos need to have a fixed aspect ratio. In such cases you can customize widget actions with special strategies.

See an example of a strategy in **Listing 4**. The widget's **boundsSuggested()** method is called by the Visual Library while we are dragging the widget; it's passed both the original and current bounds. By returning a freshly computed **Rectangle** we can override the default settings. The code first gets the image's aspect ratio and then computes the height from the new width or vice versa. This calculation takes into account the widget's borders: if we draw a fixed-size border around the photo its thickness must not affect the aspect ratio calculation.

In blueMarine, preview images are wrapped by a **Thumbnail** class. Similarly, in your applications you'll usually have a specific class containing the data you want to render in the widget. The problem is how to bind a widget to a data model. While the simplest solution appears to be to create specific getter/setter methods in **ThumbnailWidget**, this would introduce specific dependencies and require explicit class casts (for example, as the method **boundSuggested()** is general, it deals with a **Widget** rather than with my **ThumbnailWidget**). Instead, I've used a **Lookup**, a very useful class from the NetBeans Platform (which is also available for use in plain Swing projects). It acts as a container of custom objects, which can be retrieved by specifying their class name. In **Listing 4**, you see that the thumbnail is retrieved by **getLookup().lookup(Thumbnail.class)**. When we discuss the **ThumbnailWidget** class we'll see how the **Thumbnail** object was made available.

Listing 5 shows the code for bringing the widget to the front. This is an example of how you can define new actions. I've extended the **WidgetAction.Adapter** class, which gets invoked by mouse and keyboard listeners, and overridden the relevant method.

Actions are bound to the widget by defining a "pipeline" to which mouse and keyboard events are delivered. Sometimes an event is propagated through the whole pipeline; in other cases a certain action consumes it definitely. The propagation of events is controlled by returning some specific flags such as **State.CONSUMED** (which stops the propagation).

The ThumbnailWidget

Now it's time to take a look at the widget's code. While the Visual Library already provides an **ImageWidget** class which renders a generic **Image**, we need something more complex, for the following reasons:

• First and most important from a performance perspective, reading an image from a file needs some time, and a Light Table can contain tens of images. For instance, twenty images requiring 50ms each would lead to a full second of loading time. We can't spend all this inside the event thread, or the Light Table would be sluggish. blueMarine deals with this by means of a **ThumbnailRenderer** class that manages image loading on demand and renders placeholders while images are not ready.

• Also, blueMarine will soon support image manipulation, and I'll need to update all representations in real time when such changes happen (e.g. by painting specific decorations when a thumbnail is not up to date). This is solved using the **Node** class's capability of firing events that notify updates – and then **ThumbnailRenderer** will do all the required work.

Listing 5. Customized action for bringing a widget to the front with a mouse click

```
private static final WidgetAction.Adapter
    bringToFrontAction = new WidgetAction.Adapter()
{
  @Override
  public State mouseClicked (final Widget widget,
      final WidgetMouseEvent event)
  {
    if (event.getButton() == MouseEvent.BUTTON1) {
      widget.bringToFront();
      return State.CONSUMED;
    }
    return State.REJECTED;
  }
};
```

platform.netbeans.org/tutorials/60/nbm-visual_library.html
A good starting point for learning the Visual Library

bluemarine.tidalwave.it
The blueMarine Project

Listing 6. The ThumbnailWidget

```
public class ThumbnailWidget extends Widget {
  private final Node node;
  private final Thumbnail thumbnail;
  private ThumbnailRenderer thumbnailRenderer = DEFAULT_THUMBNAIL_RENDERER;
  private final Lookup lookup;

  private final PopupMenuProvider popupMenuProvider = new PopupMenuProvider() {
    public JPopupMenu getPopupMenu (final Widget widget, final Point location) {
      return node.getContextMenu();
    }
  };

  public ThumbnailWidget (final Scene scene, final Node node, Dimension size) {
    super(scene);
    this.node = node;
    thumbnail = thumbnailManager.findThumbnail(node.getLookup().lookup((DataObject.class)));
    size = new Dimension(size);
    lookup = new ProxyLookup(node.getLookup(), Lookups.fixed(node, thumbnail));
    final BufferedImage image = thumbnail.getImage();

    if (image != null) {
      final int width = image.getWidth();
      final int height = image.getHeight();
      final double hScale = size.getWidth() / (float)width;
      final double vScale = size.getHeight() / (float)height;
      final double scale = Math.min(hScale, vScale);
      size.setSize(Math.round(scale * width), Math.round(scale * height));
    }

    setUnselectedBorder(DEFAULT_UNSELECTED_BORDER);
    setSelectedBorder(DEFAULT_SELECTED_BORDER);
    setBorder(unselectedBorder);
    final Insets insets = getBorder().getInsets();
    size.width += insets.left + insets.right;
    size.height += insets.bottom + insets.top;
    setPreferredSize(size);
    setMinimumSize(DEFAULT_MINIMUM_SIZE);
    getActions().addAction(ActionFactory.createPopupMenuAction(popupMenuProvider));
  }

  @Override
  public Lookup getLookup() {
    return lookup;
  }
  ...
}
```

• Lastly, it's necessary to implement a context menu for the widgets, which must be coherent with the rest of the application. **Node**s again provide support for this. (See the result in **Figure 3**).

Considering all this, it seems obvious that we need to implement a special widget class that delegates the implementation of context menus to the **Node** class and the rendering operations to **ThumbnailRenderer**. Let's first concentrate on the widget's creation.

In the code shown in **Listing 6**, I set some fields to refer to the **Node** and **Thumbnail**; then I adjust the user-specified size to comply with the photo's aspect ratio. I've previously mentioned the role of the **Lookup** class in linking a widget to its model, and shown how to extract the model from a properly prepared **Lookup** instance. Now, in **Listing 6**, you can see how the **Lookup** instance is prepared. A **ProxyLookup** is a NetBeans Platform class that "merges" two existing instances of **Lookup** – the one coming from the **Node** (required for the context menus to work) and a new one that contains both the **Node** and the **Thumbnail**.

To paint a custom widget, we add code to the **paintWidget()** method (see **Listing 7**). The obvious part here is that the image rendering is delegated to my **thumbnailRenderer.paint()** method. Less trivial is managing the scaling (remember, a scene can be zoomed in and out).

This is done by controlling the scale of **Graphics2D**.

Listening for node changes

Changes in the representation of photos are handled by firing events on the relevant **Node**. So that our **ThumbnailWidget** updates correctly, we need to setup a **NodeListener**, which you can attach and detach in the **notifyAdded()** and **notifyRemoved()** methods (see **Listing 8**). These are called when the widget is added to or removed from a scene (you can think of them as a kind of life-cycle control).

Now take a look at the **doRepaint()** method in **Listing 9**. Notice that it must cope with the usual Swing threading issues, since **Node** events can be fired by an arbitrary thread. Also, after a widget has been changed (in this case by calling its **repaint()** method), the scene must be validated. Otherwise you won't see any updates.

Handling borders

The cream on the cake is adding visual cues to the widgets. We want to render different borders around the photos according to their selection state: no border for unselected widgets, a white border if selected, and a special "resize border" when you hover over the photo (see **Figure 4**).

First, we need to override the **notifyStateChanged()** method, which is called whenever the widget changes state (see **Listing 10**). It receives two parameters representing the old and the new state. We use the **isSelected()** and **isHovered()** methods to choose the proper border.

There's a subtle problem here: borders can vary in thickness. By default the Visual Library preserves the overall size of a widget,

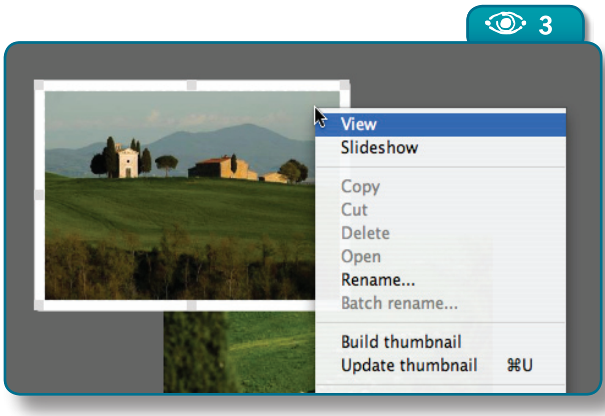


Figure 3. Context menu for a widget

Listing 7. Rendering the custom widget

```
@Override
protected void paintWidget(){
  final Graphics2D g = getGraphics();
  final AffineTransform transformSave = g.getTransform();
  final Rectangle bounds = getClientArea();

  thumbnailRenderer.setThumbnail(thumbnail);
  g.translate(bounds.x + 1, bounds.y + 1);
  bounds.width -= 1;
  bounds.height -= 1;
  thumbnailRenderer.setBounds(bounds);
  final double zoomFactor = getScene().getZoomFactor();
  g.scale(1 / zoomFactor, 1 / zoomFactor);
  thumbnailRenderer.paint(g);
  g.setTransform(transformSave);
}
```

Listing 8. Listening for changes

```
private final NodeListener iconChangeListener =
   new NodeAdapter()
{
  @Override
  public void propertyChange(final PropertyChangeEvent event) {
    if (Node.PROP_ICON.equals(event.getPropertyName())) {
       doRepaint();
    }
  }
};

@Override
protected void notifyAdded() {
  super.notifyAdded();
  node.addNodeListener(iconChangeListener);
}

@Override
protected void notifyRemoved() {
  super.notifyRemoved();
  node.removeNodeListener(iconChangeListener);
}
```

Listing 9. Forcing the repaint of a widget

```
private void doRepaint() {
  //The Nodes API can fire events outside the AWT Thread
  if (SwingUtilities.isEventDispatchThread()) {
    repaint();
    getScene().validate();
    //required or repaint() doesn't work
  }
  else {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        repaint();
        getScene().validate();
      }
    });
  }
}
```

Listing 10. Reacting on widget state changes

```
@Override
protected void notifyStateChanged(final ObjectState
    oldState, final ObjectState newState)
{
  super.notifyStateChanged(oldState, newState);
  final boolean isResizableBorder =
    newState.isHovered();
  final Dimension size = getPreferredSize();
  final Insets o = getBorder().getInsets();

  setBorder(newState.isSelected()
    ? (isResizableBorder ? resizeSelectedBorder :
        selectedBorder)
    : (isResizableBorder ? resizeBorder :
        unselectedBorder));

  // Preserve client area size
  if (size != null) // null at initialization {
     final Insets n = getBorder().getInsets();
     size.width += n.left + n.right - o.left - o.right;
     size.height += n.top + n.bottom -
       o.top - o.bottom;
     setPreferredSize(size);
  }
}
```

Listing 11. Widget Borders

```
private static final int BORDER_THICKNESS = 8;
private static final Color NORMAL_COLOR =
   new Color(200, 200, 200);
private static final Color NORMAL_GLOW_COLOR =
   new Color(200, 200, 200, 100);
private static final Color SELECTION_COLOR =
   new Color(255, 255, 255);
private static final Color SELECTION_GLOW_COLOR =
   new Color(255, 255, 255, 128);
private static final Color RESIZE_COLOR =
    new Color(220, 220, 220);

private static final Border DEFAULT_UNSELECTED_BORDER =
   BorderFactory.createRoundedBorder(
      BORDER_THICKNESS, BORDER_THICKNESS, NORMAL_COLOR,
        NORMAL_GLOW_COLOR);
private static final Border DEFAULT_SELECTED_BORDER =
   BorderFactory.createRoundedBorder(
       BORDER_THICKNESS, BORDER_THICKNESS,
         SELECTION_COLOR, SELECTION_GLOW_COLOR);
```

so setting a different border thickness would change the space reserved for the photo. To preserve the size of the photos, we just need to compute the change in the border and adjust the widget size (also in **Listing 10**).

Some final words about borders. The **Border** class for **Widget** is different from the usual Swing **Border** classes (see **Listing 11**). Widget borders are more complex. Also, there's a similar **BorderFactory** which provides some preset borders useful in most cases. In the Light Table, the default borders are just rectangles with rounded corners that can be created with **BorderFactory.createRoundedBorder()**. You can create several common borders similarly: for instance, **BorderFactory.createResizeBorder()** gives you a standard "resize border" that is painted as a dashed line with "control handles". In some special cases, we can write code to define customized borders. For instance, **Listing 12** shows how to implement a "compound border" which sticks two borders together.

Conclusions

In this article, we've seen many features of the Visual Library and a "creative use" for it that goes a little outside its most common scope. This provides us some examples for understanding how the library can be extended to comply with your needs. We've only scratched the surface, however. For instance, we didn't explore connection widgets, which are another powerful feature. But that would be material for another article!

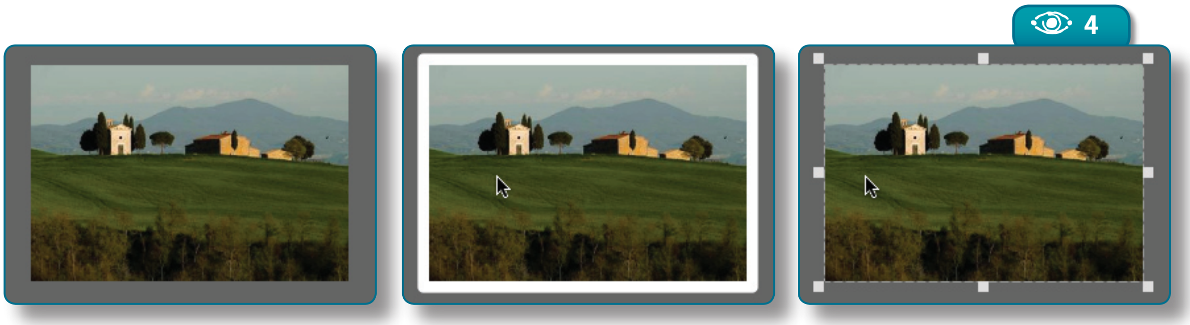


Figure 4. Visual cues for photos in the Light Table (normal, selected and resizing)

Listing 12. Creating a custom, compound Border

```
import org.netbeans.api.visual.border.Border;

public class CompoundBorder implements Border {
  private final Border border1;
  private final Border border2;

  public CompoundBorder (final Border border1, final Border border2) {
    this.border1 = border1;
    this.border2 = border2;
  }

  public Insets getInsets() {
    final Insets i1 = border1.getInsets();
    final Insets i2 = border2.getInsets();
    return new Insets(Math.max(i1.top, i2.top), Math.max(i1.left, i2.left),
        Math.max(i1.bottom, i2.bottom), Math.max(i1.right, i2.right));
  }

  public void paint (final Graphics2D g, final Rectangle bounds)  {
    // You should actually check if the insets are different...
    border1.paint(g, bounds);
    border2.paint(g, bounds);
  }

  public boolean isOpaque() {
    return border1.isOpaque() || border2.isOpaque();
  }
}
```

A new, smarter editor, the next generation of the ground-breaking Matisse GUI builder, Ruby support
NETBEANS : The only IDE you need.

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

NetBeans magazine