

NetBeans

November . 2006

magazine

Java EE 5 in Practice

Exploring Java EE 5 with NetBeans 5.5

Not Just an IDE

Modular development with the NetBeans Platform

UML Modeling in Action

Using the NetBeans UML Modeling Project

BPEL Designer

Creating BPEL processes and composite applications

Plug in to Google

Create a plug-in module with 3 lines of code

Java SE Web Services

Developing web services productively using the IDE

NetBeans Worldwide

A global sampling of the NetBeans community

Boosting Enterprise Productivity



NetBeans
magazine

Publisher & Editor-in-Chief

Leonardo Galvão
leonardo.galvao@gmail.com.br

Assistant Editor

Oswaldo Doederlein
opinalli@gmail.com

Design and Layout

pH Design (phdesign.com.br)



Graphic Designers

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

Illustrator

Felipe Machado

Contributors

Fernando Lozano
Geertjan Wielenga
Leonardo Galvão
Milan Kuchtiak
Sherry Barkodar
Oswaldo Doederlein
Tim Boudreau

Community Support

Robert Demmer
Geertjan Wielenga
Janice Campbel
John Jullion-Ceccarelli
Roman Strobl

*NetBeans Magazine is
supported by NetBeans.org*

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

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

Enterprise Success

Keeping its tradition of offering seamless integration and extensive functionality out-of-the-box, NetBeans reaches its second major release in 2006. It took the developer team just six months to create a product that impresses by the sheer amount of new features it aggregates. Take the new Enterprise Pack, for example. You have features ranging from business process modeling to secure web services, to XML schema creation. Add to that a complete UML modeler, plus the full Java EE 5 support and other enhancements in the core IDE... and there's a whole new NetBeans world to explore.

This edition of NetBeans Magazine covers a range of new features in NetBeans 5.5 and its main extensions while also delving into core Java technologies. A thorough tutorial puts the new features in Java EE 5 through their paces, using NetBeans to build a complete enterprise application with JPA persistence, EJB 3.0 components and JSF. In it you see the way NetBeans IDE boosts productivity in enterprise development, and how the new features in Java EE measure up in real-world scenarios. In two other articles, you learn step by step how to use the new visual BPEL designer to create and test a business process, and also how to develop web services starting from Java SE projects.

Module development is one of NetBeans' strongest points, and in this edition a quick-start tutorial shows you how to develop a very useful plug-in module that connects to Google Code Search (just add three lines of code and NetBeans does the rest). And a comprehensive article about the NetBeans Platform presents the design ideas behind NetBeans' pluggable architecture. You see how to use the Platform to slash development time while improving software quality. Also watch modular development in action in an application that pushes the concept to the limit.

UML modeling is covered in a novel way. Instead of learning each feature mechanically, you'll see the modeling functionality at work by reverse-engineering a complete Java application. Starting from a non-visual model, the tutorial builds a set of UML diagrams that show all the relevant structure and behavior of an example app.

Finally, in our opening section, we highlight the pervasiveness of the NetBeans community, by showcasing a sampling of NetBeans-related projects and initiatives from around the world. NetBeans adoption is expanding on all continents, and the community is alive everywhere.

Happy coding!

Leonardo Galvão



A Long Way

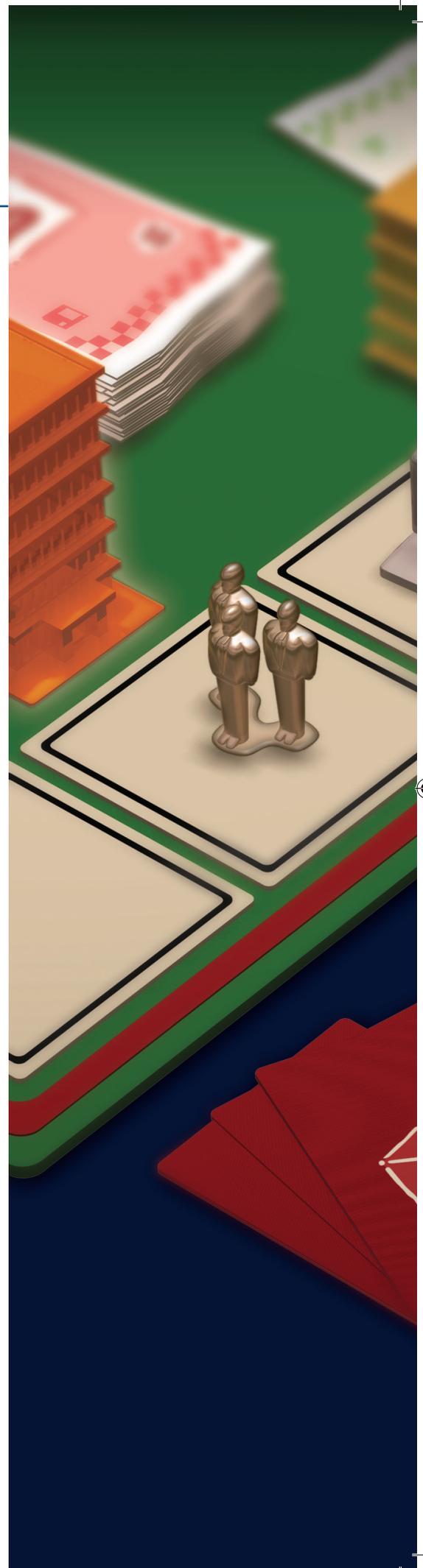
NetBeans has come a long way. When a handful of students, back in 1996, started the project that was to become NetBeans, did they know where it would all end? Aiming to write a Delphi-like Java IDE in Java, they began the long journey that has now come to NetBeans IDE 5.5. However, the most significant infrastructural change came in the 4.0 release. The old “mount” system was replaced by a new project system, fully based on Ant. Project tasks were carried out by Ant scripts, making them easily extendable and freeing applications from lock-in in the IDE. The window system was thrown out of... the window. It was rewritten, creating a state-of-the-art system that was a joy to behold, and smooth and easy to work with.

Now, with the release of NetBeans IDE 5.5, another giant leap has been taken. Initially aiming to provide “only” full Java EE 5 support, the 5.5 release now also provides “packs” that integrate the crown jewels of several other Sun tools into NetBeans IDE. For example, together with the 5.5 release, we also have the final release of the SOA-oriented Enterprise Pack, a preview version of the C/C++ Pack, and the Visual Web Pack, which brings an award-winning set of visual tools for web development. On top of that, UML modeling is supported and mobile developers have the Mobility Pack for CLDC and Beta 2 of the Mobility Pack for CDC.

But, significantly, NetBeans is more than a set of tools. It is also a community, a community which has grown exponentially over the past years. Many have testified that the breadth of the package offered by NetBeans IDE is unsurpassed. And now, while acquainting ourselves with the NetBeans IDE 5.5 release, we look forward to NetBeans IDE 6.0. The good rumors are many and the tension builds. Read through this magazine and realize... there is even more in store in the coming period. NetBeans has a lot to live up to and a growing community cheering it on.



Geertjan Wielenga



Contents

05

NetBeans Community Worldwide

LEONARDO GALVÃO

A global sample of the sprawling NetBeans community: people, projects and places

08

Not Just an IDE

TIM BOUDREAU

Working with the NetBeans Platform and the NetBeans Module System – and the case for modular development.

18

Plug in to Google Code Search

GEERTJAN WIELENGA

Create a plug-in module to integrate Google Code Search in NetBeans, with only three lines of Java code

22

Java EE 5 in Practice

OSVALDO DOEDERLEIN

A critical exploration of Java EE 5's new productivity features, focusing on EJB 3.0, JPA, and NetBeans 5.5 tooling support

Java SE Web Services

MILAN KUCHTIK

Develop, consume and deploy Java SE based web services productively using the NetBeans IDE

36

UML Modeling in Action

FERNANDO LOZANO

Learn to use the NetBeans UML Modeling Project by reverse engineering a Java application in the IDE

40

Strategic Partners

MATT VOLPI

Companies adding new features and capabilities to NetBeans amplify the value to developers

54

BPEL Designer Kick Start

SHERRY BARKODAR

Exploring visual design capabilities in NetBeans to create, test and deploy a BPEL process and a composite application

56

NetBeans Community Worldwide

NetBeans has never been more global. The NetBeans community is growing at a fast pace in countries on six continents. And with a number of ongoing localization projects, the IDE and Platform are ever closer to developers around the world. Here we show a sampling of international NetBeans-related projects and initiatives, with special focus on geographical distribution.



The maps in this section are stylized and abstract, not reflecting exact geographic locations and country limits.

France/Belgium

The popular French-language *developpez.net* portal hosts a very active NetBeans forum and FAQ. This volunteer-driven developer website counts among its members many Java champions, and is a reference in the French-speaking world, being featured as a Top-50 JUG at Java.net.

Germany

The recent NetBeans User Group Munich meeting brought international NetBeans evangelists and local NetBeans specialists to Munich, Germany, where they delivered presentations to a packed audience (why not create a NUG in your country too?). Also check out the Munich NUG members' Software Quality Environment project (sqe.dev.java.net), which provides NetBeans integration for software quality tools such as PMD, FindBugs and Checkstyle.

Czech Republic



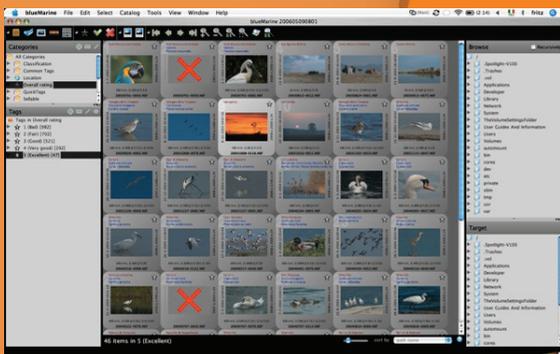
Prague, the Czech capital is the birthplace of NetBeans and home to the core NetBeans development team. Many evangelists and executives have adopted the city as their home. The Mobility Pack is also developed here.

Kosovo

Kosovo, a small province governed by a United Nations mission, will soon have its localized NetBeans version, with translation to Albanian moving fast. And the Kosovo developer community is already creating national-scale projects using the NetBeans IDE.

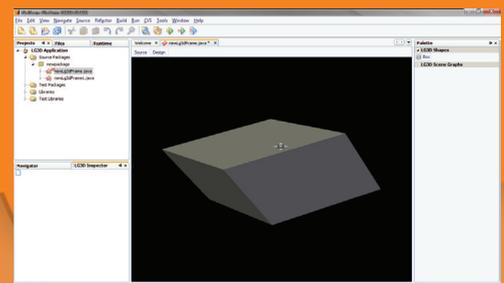
Italy

The Italian Blue Marine application (bluemarine.tidalwave.it) is being ported to the NetBeans Platform. Providing features for managing the digital photography workflow, this beautiful open-source Java application goes beyond photo organization tools, and is very extensible due to its modular design.



Egypt

In Egypt, the Plethora project, due out later this year, will integrate NetBeans with Project Looking Glass, the open-source 3D platform built with Java 3D. The plug-in module, which is being developed by a group of students in the American University in Cairo, will provide code generation and a form editor.



U.S.A.

While the core NetBeans development is done in Prague, the lion's share of development for the Visual Web Pack and the Enterprise and C/C++ Packs is done in the United States. And this is where the NetBeans World Tour kicks off every year, at JavaOne, San Francisco.

Panama

The Panama developer community has been actively adopting NetBeans as their main Java IDE, using it both in university and commercial projects. A sharp and up-to-date blog in Spanish, *avbravo.blogspot.com*, highlights all things NetBeans.

Brazil

Brazilian Portuguese has been chosen as an additional "official language" for NetBeans (the others are English, Simplified Chinese, and Japanese). By the time you read this, the Brazilian community, with support from Sun and SouJava, will be putting the final touches on the translations. The developer community in Brazil is very active, having started several successful NetBeans projects, such as GreenBox (*greenbox.dev.java.net*), a plug-in module and framework that generates code based on Spring, Hibernate, and JSF.

Azerbaijan

In Azerbaijan, a large UNESCO-sponsored Java project is being developed on the NetBeans Platform. The Java CDIS/ISIS project will allow users to query UNESCO's text retrieval system, used for bibliographic and other databases throughout the world, especially in developing countries.

Russia

Russia was the first country to have a community-only localized release of NetBeans 5.0. The leader of the Russian translation effort also co-founded the fast-moving NetBeans Translation Project (*translatedfiles.netbeans.org*). You can bring NetBeans even closer to your community by helping out with the many ongoing localization projects, or by creating a new one for your language.

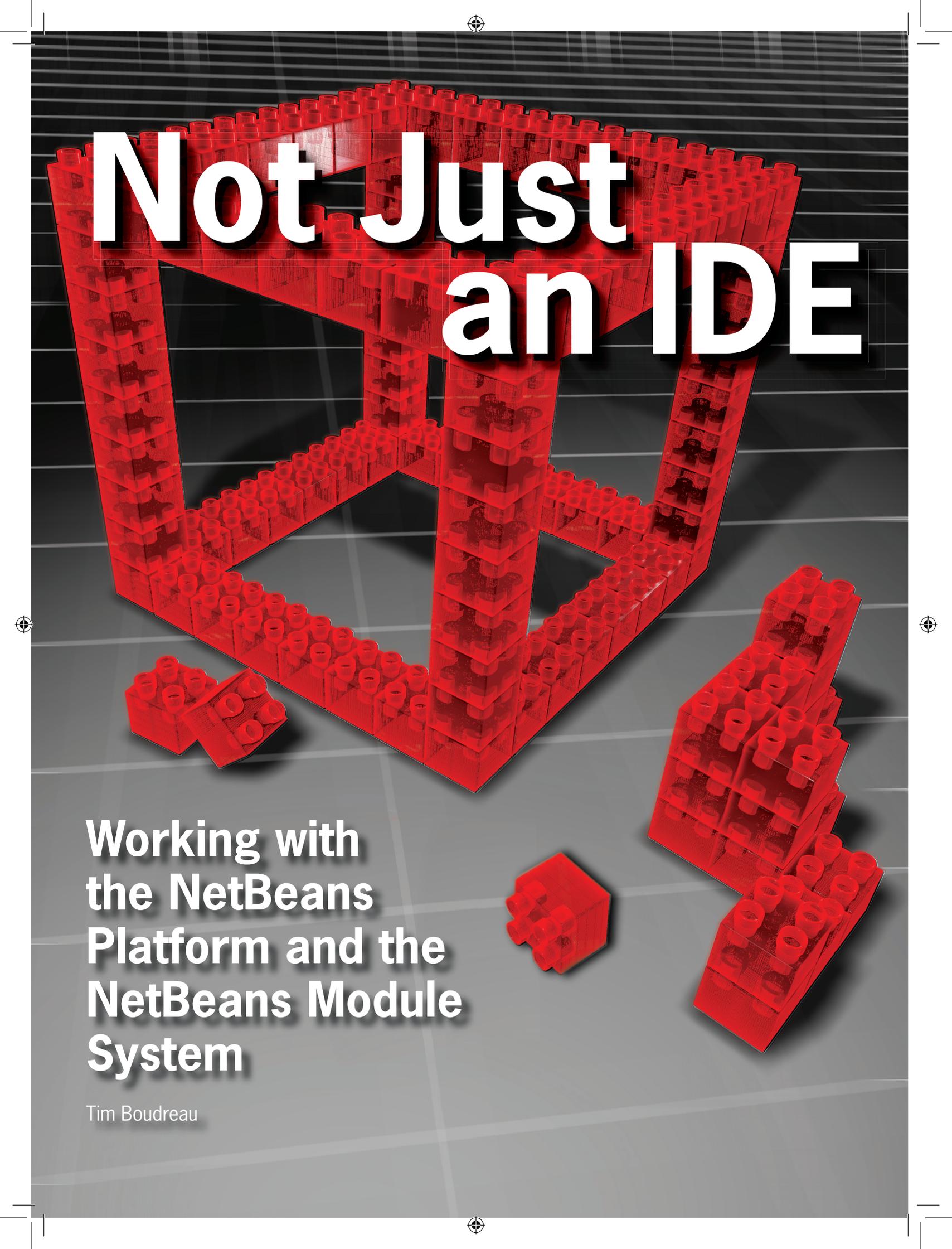
China

The BlogTrader Platform (formerly Humai Trader – *humaitrader.dev.java.net*), currently being developed in Beijing, China, is a standalone open-source application built on top of the NetBeans Platform, with source code organized as a NetBeans module-suite project. The tool provides features to track and analyze the stock market, both from historical data and real-time feeds.



Japan

Japanese speakers have an official all-Japanese NetBeans community at *ja.netbeans.org*. Community members can contribute their own content to the site and translate web content from *netbeans.org*. Sun provides support in reviewing, coordination, and CVS commits.



Not Just an IDE

**Working with
the NetBeans
Platform and the
NetBeans Module
System**

Tim Boudreau

This article covers the basics of NetBeans module development and using NetBeans Module System and NetBeans Platform, and makes a general case for modular software development as a methodology for improving quality and productivity.

You're probably familiar with the NetBeans IDE as a development tool. It is also a platform for building modular applications – even non-GUI applications. The NetBeans Platform is the underpinnings of the IDE. Think of it as a “generic desktop application”. Underneath the NetBeans Platform sits the NetBeans Module System – the engine that launches NetBeans, discovers components dynamically and resolves dependencies between modules. It's the NetBeans Module System that makes it possible to download new modules and hot-deploy/redeploy modules at runtime.

A module is simply a JAR file with some manifest entries that allow the NetBeans Module System to recognize it. All of the APIs you can write to in NetBeans live inside modules. These are divided along functional lines. So, for example, if you want to change the status bar text at runtime, you will use a class called **StatusDisplayer**. **StatusDisplayer** is part of the *UI Utilities API*. That API lives in a module (a JAR file) which your code can depend on. Building on the NetBeans Platform is essentially writing code that interacts with these APIs, and these APIs are implemented in modules just like the ones you will create.

The case for modular development

Dynamically linking applications at

runtime is an age-old problem. The historical solution is the familiar one: linking to libraries in native applications – or finding JAR files on the classpath in Java applications. These approaches work but have some drawbacks. First, there is no version management. An application linking itself together at runtime has no idea what versions of the libraries it needs. It also has no way to know if it is linking with a version which is old or incompatible (perhaps you have heard the term “DLL Hell”). Second, this scenario covers using libraries at runtime, but it makes no provision for an application which is truly assembled at runtime – one which discovers its libraries on the fly, and may want to unload and update such libraries.

The NetBeans Module System solves these problems. It is a runtime container specifically designed for applications which discover (and can update) their components at runtime. And it specifically handles inter-library dependencies very explicitly – so the application cannot be started in a state where its dependencies are not satisfied.

Of course, not everyone is writing an application that *needs* to assemble itself at runtime. My point here, though, is that the NetBeans Module System may be relevant to you even if you are not writing such an application. There are benefits to modular development that go well beyond the capabilities of a runtime container.

Consider how NetBeans itself is developed: by a community of hundreds of people that spans several continents, time zones, and teams. Our experience is that there are extraordinary benefits to modular development unrelated to the runtime capabilities of the NetBeans Module System. They have much more to do with engineering culture and sustainability of the product. In other words, these are benefits that may not apply to a one-off bit of coding, but show up over time; benefits that improve the probability of shipping a 2.0 after a 1.0, a 3.0 after the 2.0 and so forth. See more about these benefits below.

Avoiding code-entropy

Many projects start out well designed, and that design gradually decays as features are added. The system becomes more tightly coupled as expedient implementation of features creates new dependencies between parts of the application. This means it gets harder and harder to fix bugs over time. Because as the code

 platform.netbeans.org

The NetBeans Platform Web site is the main entry point for NetBeans development on the web, linking to all other development documentation. The tutorials section (*platform.netbeans.org/tutorials*) is particularly useful.

 netbeans.org/download/dev/javadoc

The NetBeans API Documentation: a comprehensive set of all documentation for all classes in public APIs in NetBeans.

The NetBeans Developer FAQ: a comprehensive knowledge base of NetBeans development information. In addition to frequently asked questions, it contains many tips, tricks and otherwise hard-to-find information, along with simple descriptions of common API classes and where to start to do various tasks. Since it is a wiki, if you get an answer to a question that is useful to you, you are encouraged to add it to the FAQ.

 wiki.netbeans.info/wiki/view/NetBeansDeveloperFAQ

becomes more coupled, fixing a bug in one place is more likely to create a bug somewhere else.

Enforcing API design

A module in the NetBeans Module System cannot reference classes in another module without declaring a dependency on that other module, *and that other module agreeing that the classes referenced are ones that are actual API*. A NetBeans module by default has no API – other modules can't see its classes. Modules that have APIs specifically say what packages they are exporting. Other modules can only reference classes from other modules if those classes are in packages the other module says it exports.

In practice what this means is that contracts between components in the system must be planned out, not just developed in an ad-hoc way. While ad-hoc development may be slightly faster, over the long haul, when you factor in the cost of maintaining that code (usually this is much more expensive than writing it in the first place), it pays off to have a system that makes it hard to write dirty hacks and encourages you to create good contracts between components.

Design hygiene

Since a module can hide most of its packages, this has the side effect that you can essentially have *public classes that are only public within the JAR they live in*. Think about this for a minute – how many times have you piled an enormous number of unrelated classes into one package, so those classes could use package-privateness to hide their implementation details from the outside world? If you can have public non-API classes (classes that can only be seen inside their own JAR), you can write *much* cleaner APIs – even provide packages that are pure API. Any API designed in this way has a better chance of being understandable to the people who will have to use it.

Comfort in deployment

With the NetBeans Module System you know that your application can never be started if it cannot satisfy its dependencies. This translates into greater peace of mind when you release a new version of your software. Of course, you'll want to make it easy for a user to satisfy those dependencies, and hopefully you're shipping something that is compatible with itself. But in terms of upgrades, not to mention nasty situations where some other application has put, say, a

buggy XML parser on all application's boot classpath, you know ahead of time that your application will not have a problem.

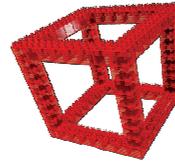
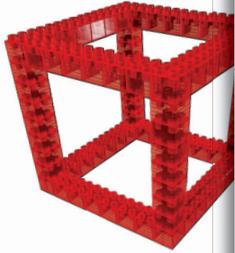
Why build on NetBeans?

Whether or not you are convinced of the benefits of modular development in general, the NetBeans Platform can save a huge amount of time for anyone developing a Swing application. In any desktop application, there is a certain amount of “plumbing” that must be written, such as wiring up menu items and toolbar buttons to actions, persisting settings, doing window management and data presentation, accessing resources, saving state on shutdown, and much more. None of these things are the business logic of the application – they are the overhead of writing a desktop app. This is stuff that is neither fun nor interesting to write, and can additionally be a source of bugs that drains off development time better spent working on logic that is the meat and potatoes of what the application *does*.

With the NetBeans Platform, you get to reuse all of this logic – and you are reusing an implementation of it that has years of engineering and quality assurance invested in it. The letter shown in the **sidebar** in the opposite page was not empty hyperbole (nor was it solicited). The NetBeans Platform can make the difference between wondering how you will ever get a project finished and being able to outpace competitors with the speed of it.

Getting started in module development

Getting started with NetBeans module development is easy. There is not a great



Platform Success

The lead developer of a team that had recently adopted the NetBeans Platform several years ago had the following to say, one month into migrating to the NetBeans Platform. While this was said a while ago, the Platform has only improved since then:

We estimate that we will manage to release a product by July, that is/has:

- 1. Faster time-to-market*
- 2. More features*
- 3. More stable*
- 4. Lower R&D cost*
- 5. Higher end-customer value*
- 6. More extendible*
- 7. Easier long-term maintenance*

After the July release, we believe that we can exponentially increase our features, since existing developers are now high on the learning curve and can mentor new developers joining us later. In fact, the previous gloomy outlook of supporting all the features requested by the upper-management, are now turned into optimistic enthusiasm to make the "Best Color Physics Software for the Textile industry in the world". And perhaps the "Color Physics" will be dropped when we include all the non-color related features, and maybe "Textile industry" will be dropped when we support all color-sensitive industries, such as plastics, printing, paint and food. Future will tell...

*To all people out there, who are considering using NetBeans for non-IDE applications, I can only **recommend**, the strongest, the OpenIDE Platform.*

To all the NetBeans developers, working at Sun Microsystems, privately or elsewhere, thanks for bringing to the world the most complete client-side framework I have ever seen.

deal of difference between developing modules to install in the IDE, to use in a standalone GUI application, or for use in a non-GUI application that uses the NetBeans Module System. The only difference between all these things is which other modules will be included in the application.

The NetBeans IDE contains specific support that makes it easy to build modules and applications, and then run, debug and deploy them. Also, since all of this module building support uses Ant scripts to do the work, setting up automated builds and other team support is easy – all the metadata are human readable and can be put in a version control system such as CVS or Subversion. NetBeans 5.0/5.5 contains three module templates that can be used to create a skeleton module (see **Figure 1**). These templates are:

1. **Module Project** – An ordinary single module, with build script and manifest. This type of module may be standalone or part of a suite of modules.
2. **Module Suite Project** – A container for a collection of modules

Figure 1.
Creating a new module in the NetBeans IDE

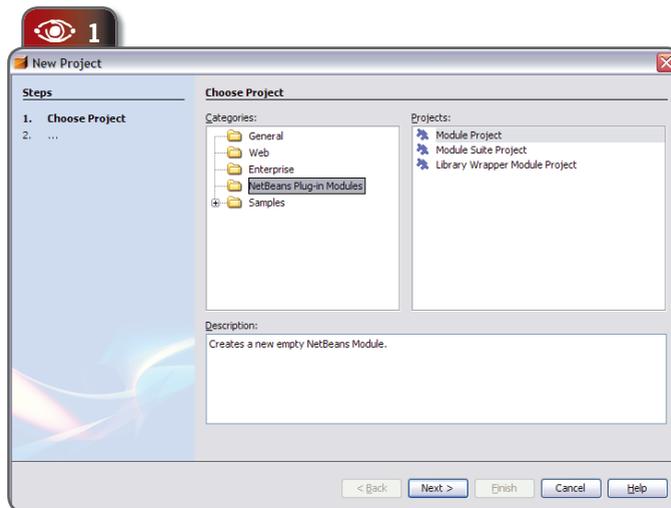
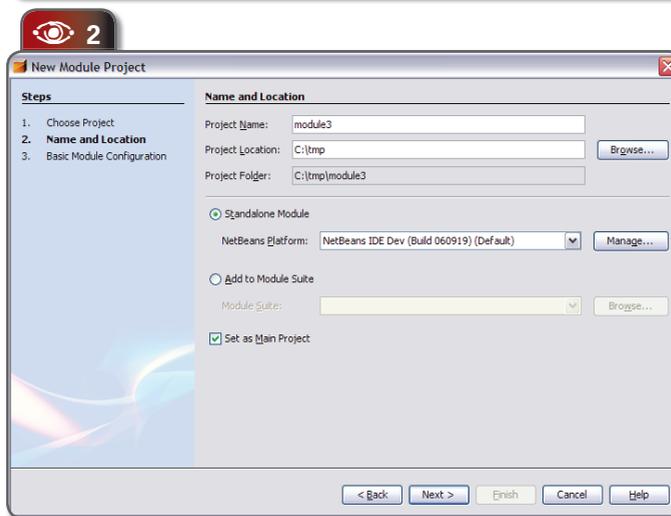


Figure 2.
Setting up the module name and location



which are deployed together, and which may have interdependencies. (Often when implementing a single unit of functionality, it is desirable to separate the implementation into separate modules which perform different functions.)

3. **Library Wrapper Module** – A wrapper module which allows you to embed any regular JAR library in the NetBeans Module System and call it. This will generate a wrapper module that contains no logic, but simply exposes the packages in the library via its manifest, so that other modules may call them.

The next step in the New Module Wizard (**Figure 2**) is very straightforward. You define where on disk to put the module that's being created, and whether or not it should be part of an existing suite of modules (you can always add it to a suite later).

After this, you simply provide some basic parameters that will identify your module (see **Figure 3**). Each module has a unique ID (Code Name Base). This is a string, and by convention should match the base Java package of your module, to avoid name collisions. If your module exposes an API, other modules will use this name to say they depend on yours and should be granted access to your module's public classes. The Module Display Name property simply provides a human-friendly name for your module.

Expressing dependencies between modules

It would not be much fun to develop modules if you needed to dig up the programmatic code name of a module in order to use its classes. The NetBeans IDE makes this easy. Your module's dependencies are

a property of your module project, which can be found in its properties dialog.

Since dependencies are between Java classes, and you are probably reading the JavaDoc documentation to find out about them, you can simply add new dependencies by class name. Just open the module properties dialog, and click *Library|Add Dependency*. You will see a dialog like the one in **Figure 4**.

Making modular development easy

The point here is that developing in a modular environment requires some additional bookkeeping. One module must say what other module it wants to use; it also can specify the minimum version it wants, etc. Such bookkeeping should not mean laborious work for the programmer. The NetBeans IDE's support for developing modules takes care of the bookkeeping quite completely, and provides a GUI for adjusting dependencies that uses the elements a developer needs to know about anyway: Java classes.

Not just for IDE plug-ins, or even GUI applications

As mentioned above, there is little difference between writing modules that enhance the NetBeans IDE, writing a GUI application or writing a non-GUI application. This too is a function of dependencies. By leaving the IDE modules out, and adding your own modules, you create a NetBeans Platform-based application – one which uses the infrastructure and GUI of the platform, without any IDE-specific appearance or functionality. By leaving out even the core UI modules and just reusing the bootstrap and module system infrastruc-

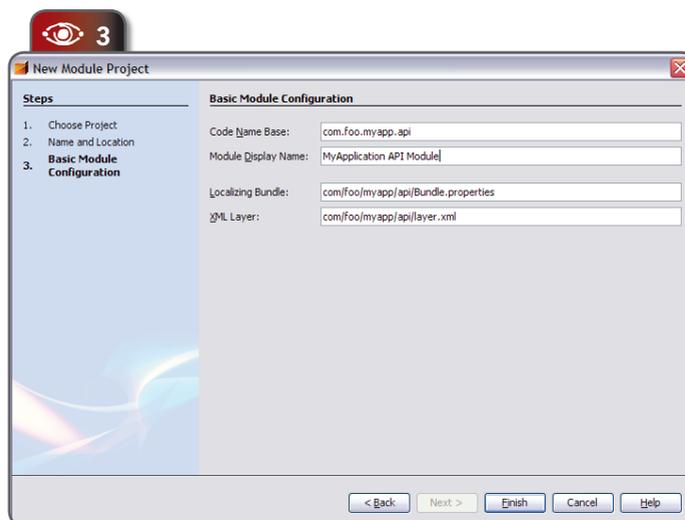


Figure 3.
Defining
module
properties

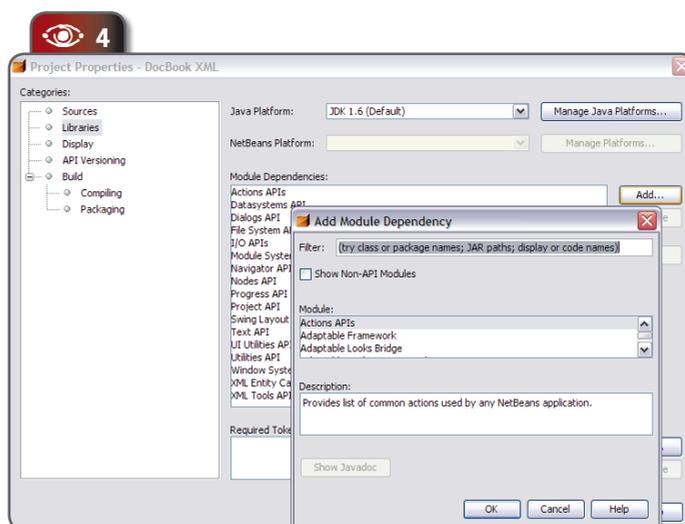


Figure 4.
Adding
dependencies

ture, you can create a modular application with no UI whatsoever.

Again, this is something that the NetBeans IDE makes simple. You can, of course, use any IDE to develop NetBeans Platform-based applications. The NetBeans IDE simply makes it easier. Creating the skeleton of an application is as simple as creating a new Module Suite using the project template shown before. Deciding whether it will be a plug-in for the IDE or a separate application is a configuration step which can be changed at any time. The UI for this is part of the properties dialog of the module suite. The first step is specifying if the suite is to be a standalone application (with its own splash screen, name and other customizations), or just a set of modules to drop into the IDE (see **Figure 5**).

The next step is deciding what modules should and should not be part of the application. By default you are building against the copy of NetBeans you are running, but you can also build against a differ-

netbeans.org/community/lists The *dev@openide.netbeans.org* mailing list is the place to ask questions – you will reach the entire NetBeans development team at Sun, plus thousands of developers who are building their own modules and NetBeans-based applications.

Figure 5.
Creating a standalone application

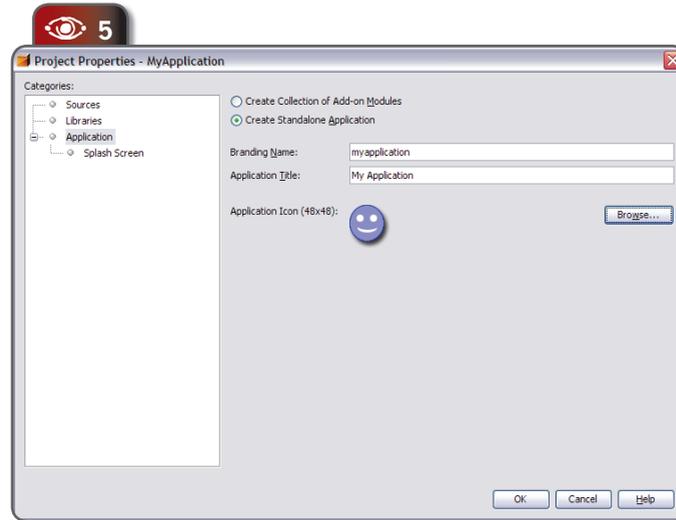


Figure 6.
Choosing components of an application

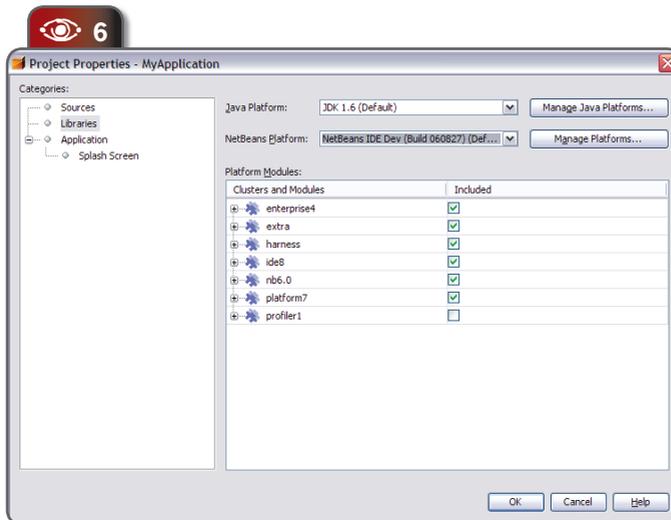


Figure 7.
Creating the Paint Application Sample App



ent copy of NetBeans, such as the plain platform distribution. The GUI for deciding what modules are used is also part of the suite properties dialog (**Figure 6**). When you generate a zip or Java Web Start

distribution of your application (one click on the suite's popup menu will do that), this GUI also determines what modules will be packaged into your distribution. The UI for this is the Libraries pane of the properties dialog.

Building the examples

NetBeans comes with some example module code built in. In the New Project Wizard, there is a category called Samples. In it you can find two sample applications built on NetBeans.

The Paint Application Sample App

In the NetBeans Module Projects category you can see a sample called Paint Application (see **Figure 7**). This is a pre-built application consisting of three modules. The wizard allows you to unpack it onto disk, build, run, debug and modify it.

The application's functionality is simple. It allows the user to draw on the screen with the mouse, and save the result as a PNG format image file. What is immediately noticeable is that, while its UI is simple, it is quite a polished application. It has a splash screen, can save settings on exit, supports editing multiple documents with drag-and-drop windows, is fully localized, has menus and toolbars that are well designed and well behaved. And the initial version of this application took all of 45 minutes to write!

If you look at the code, you will see that the entire application's logic is only *four* Java classes, and two of those are trivial **Action** implementations for creating new documents and saving (see **Figure 8**).

The Paint Application sample (see it in action in **Figure 9**) consists of three modules. They are:

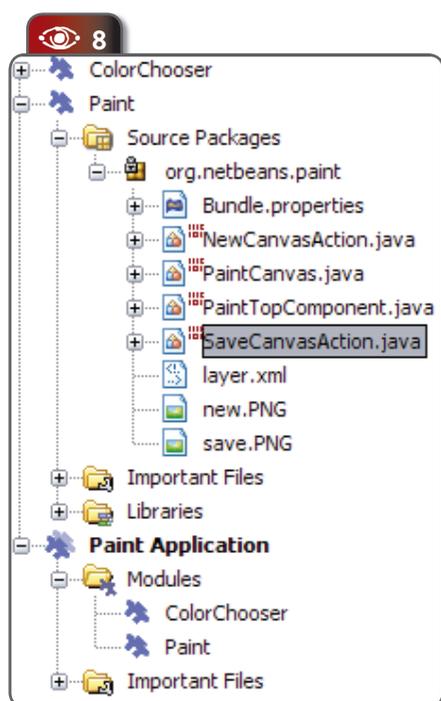


Figure 8. Structure of the Paint sample application

- **The Module Suite** – This contains no code. It simply allows the modules it contains to depend on each other and be deployed as a unit.

- **The Paint Module** – This is where the business logic lives. It contains the actions and GUI components that you see in the application's main window.

- **The ColorChooser Wrapper Module** – This is a no-code wrapper module for a popup color chooser component (available from colorchooser.dev.java.net) which is used on the toolbar in the application.

Dependencies between components are unidirectional – the Paint Module depends on the color chooser library, but the color chooser library knows nothing about the Paint module. Generally it is worth thinking about dependency relationships between pieces of a system before starting to code. Occasionally it will appear, in development, that circular dependencies (one JAR depends on a second JAR's class and

that second JAR depends on the first one). Such situations usually mean there is a third piece which is the common functionality both need, and that should be factored out into a single module both of the others depend on.

Modular development In action

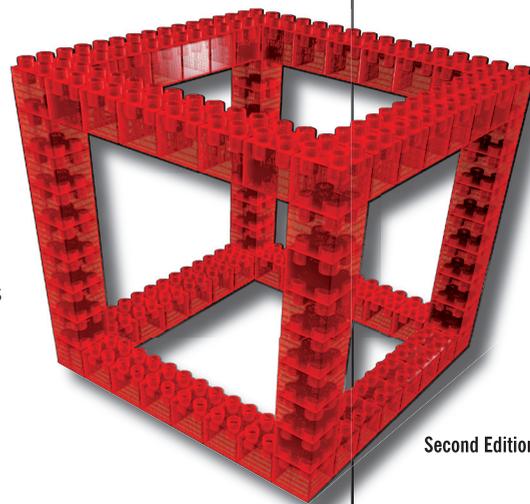
To really get a sense of the power of the modular development paradigm, and what you can do with the NetBeans Platform, there is a much enlarged and enhanced version of the original Paint Application tutorial. Its source code can be found at imagine.dev.java.net. The Imagine application takes the painting application design to an extreme of creating an application with the potential to include functionality similar to that of GIMP or Photoshop.

It starts from the same premise as the original paint application, still uses the color chooser control and has a class called **PaintCanvas** which is reminiscent of the Paint Sample Application. But it goes much farther in having a powerful user interface, and provides its own APIs to allow additional tools to be plugged in.

Imagine: a Modular Demonstration Application

Imagine is an image editor (see **Figure 10**). Much like other image editors, it uses a model in which an image is constructed of image *layers*, each of which is independently editable. It has a palette of *tools* which can be selected in order to use them to edit the image. And it has a number of windows in its UI, including a layers view, a customizer for the selected tool, a menu for effects that can be applied to a selection or a layer, the image editor itself, and an edit history browser window.

While this application is not ready to replace your favorite image editor tomorrow, it is designed for extensibility, so that features can be plugged in separately. And more importantly, it is designed so that its component parts have well defined contracts between them. It will be harder for a change in one part of the application to break another part. This is particularly useful in the case that it would be developed by people not physically in the same place. Building in an environment that encourages healthy architectural practices has benefits for the application's maintainability.



The NetBeans Source Code: while you don't need the sources to develop your own code, they can be an invaluable source of examples of how to do things right – simply find a module that does more or less what you need to do and see how it works. All sources are available via anonymous CVS: **cvs -d:pserver:anoncvs@cvs.netbeans.org/cvs standard_nowww** will download the basic sources.

Figure 9.
Running the
Paint sample
application

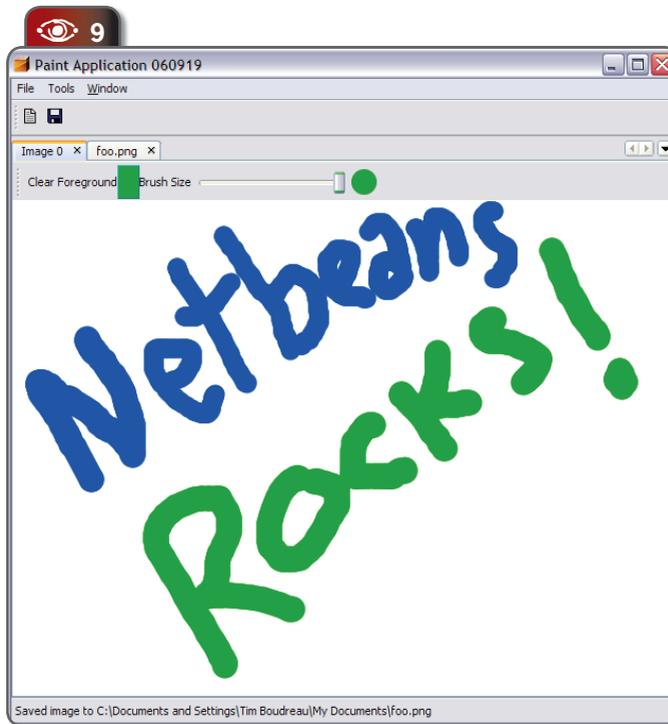
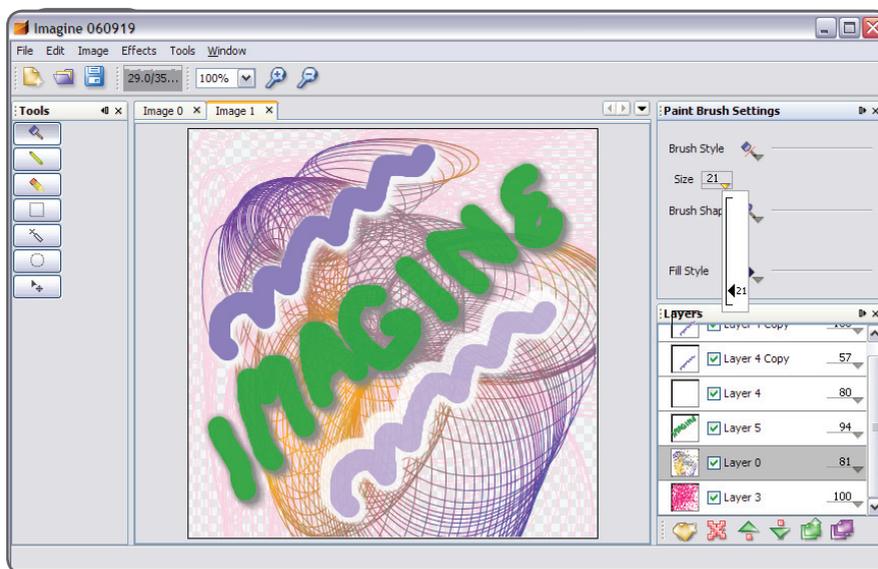


Figure 10.
Imagine in
action



The way that Imagine breaks up into modules is the interesting part. Each piece of UI you see – each window component (and, incidentally, the effects menu), is implemented in a separate module (**Figure 11**). Now, were this a production application, it might not be broken up to quite this degree, but it demonstrates the power of this approach very well. The editor in the center of the window comes from one module. It can be removed from the system and

the system will still function (though it won't do very much). More importantly, it could be replaced by a totally different editor that operated on, say, SVG – and with the exception of the raster-based tools, the rest of the application would not need to change one bit. The tools palette lives in a separate module; it contributes the selected tool to a global selection context which the editor component listens to. The tool customizer is another module – one which, like the editor, listens to the global selection and if a tool appears in it, displays its customizer. The effects menu is implemented in yet another module, which locates all registered effects and provides a UI for them. The editor window contributes the current image and its layers to the global selection; the layers window listens for this and displays the layers of the current editor. And so forth...

The truly interesting thing to notice is that *none of the modules I have just mentioned depend on (use classes from) each other*. There is a single module called Paint API which

defines Java classes such as **Tool**, **Layer**, **Layers**, and **Effect** (see **Figure 12**). Each of the other modules either provides or consumes instances of these classes; the communication mechanism is the global selection, which uses a NetBeans API class/concept called **Lookup** (which is essentially a **Map** where the keys are **Class** objects and the values are one

or more instances of the **Class** key. All of these modules simply depend on the Paint API module's classes. So any piece can be completely replaced without the other parts of the application being disturbed in the slightest.

If someone develops additional tools or effects, they can

be distributed as a new module. There is no need for recompiling the entire application. The NetBeans Platform even includes the optional Update Center module (autoupdate), which will let you deliver new modules and new versions of modules as simply as by putting some files on a web server.

Role-based deployment with Java Web Start

Where this sort of thing gets really exciting is when it comes to applications that are used by multiple types of users. The showcase for this approach (and originator of it) is Nokia NetAct – an application for managing cellular networks. The way it works is this. You have an activity which is information-based and involves many people; those people will have different roles and needs. Rather than optimize the application for one set of users or create multiple similar applications for different roles, you create one application, with different sets of modules to provide the UI for

varying roles. Then deploy it using Java Web Start (JNLP), on an application server. The users log in to a web page; since they have authenticated, the server knows who they are and what their job is. The user clicks a link that starts the application via Java Web Start. *Depending on who they are, they will be delivered a different set of modules.* The underlying APIs and business logic may be the same across all users, but the user interface pieces and what functionality users have access to will vary depending on what they need to do their job.

A similar scenario is one where an organization needs to produce many different versions of an application with slight differences. For example, imagine an application used for both individuals and businesses to calculate or pay their taxes. Much of the logic underlying either a business's or individual's work will be the same. These should be provided in modules common to the whole application. There will be differences in the user interface, and in what functionality is made available. Imagine deploying such applications, sharing all the code that should be shared between them, with no nasty hacks, no duplication, no extra testing needed, because the architecture you're building on is *designed* for this sort of scenario!

Conclusions

It is my hope that this article has whetted your appetite to learn more about the NetBeans Platform and get some of the benefits of it in your own code! ☒

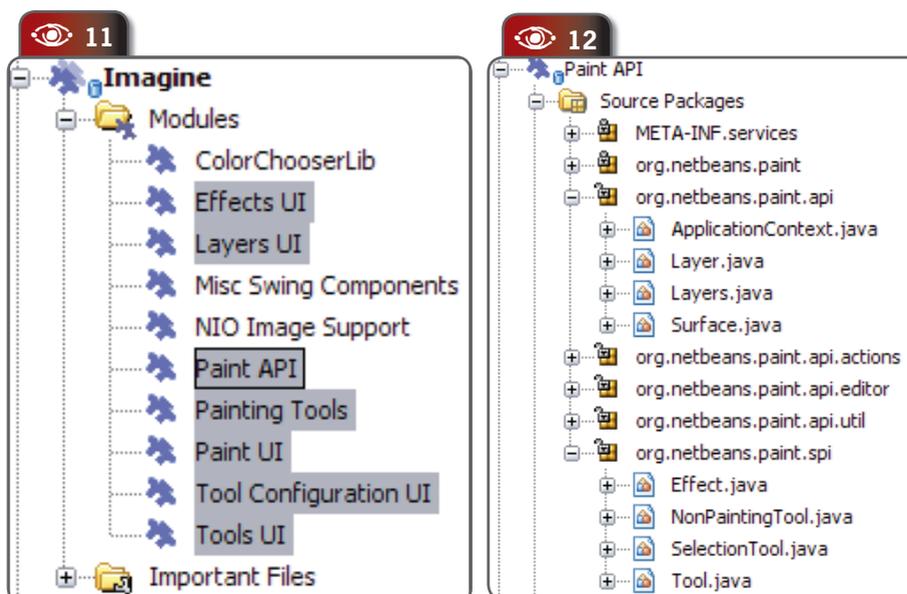
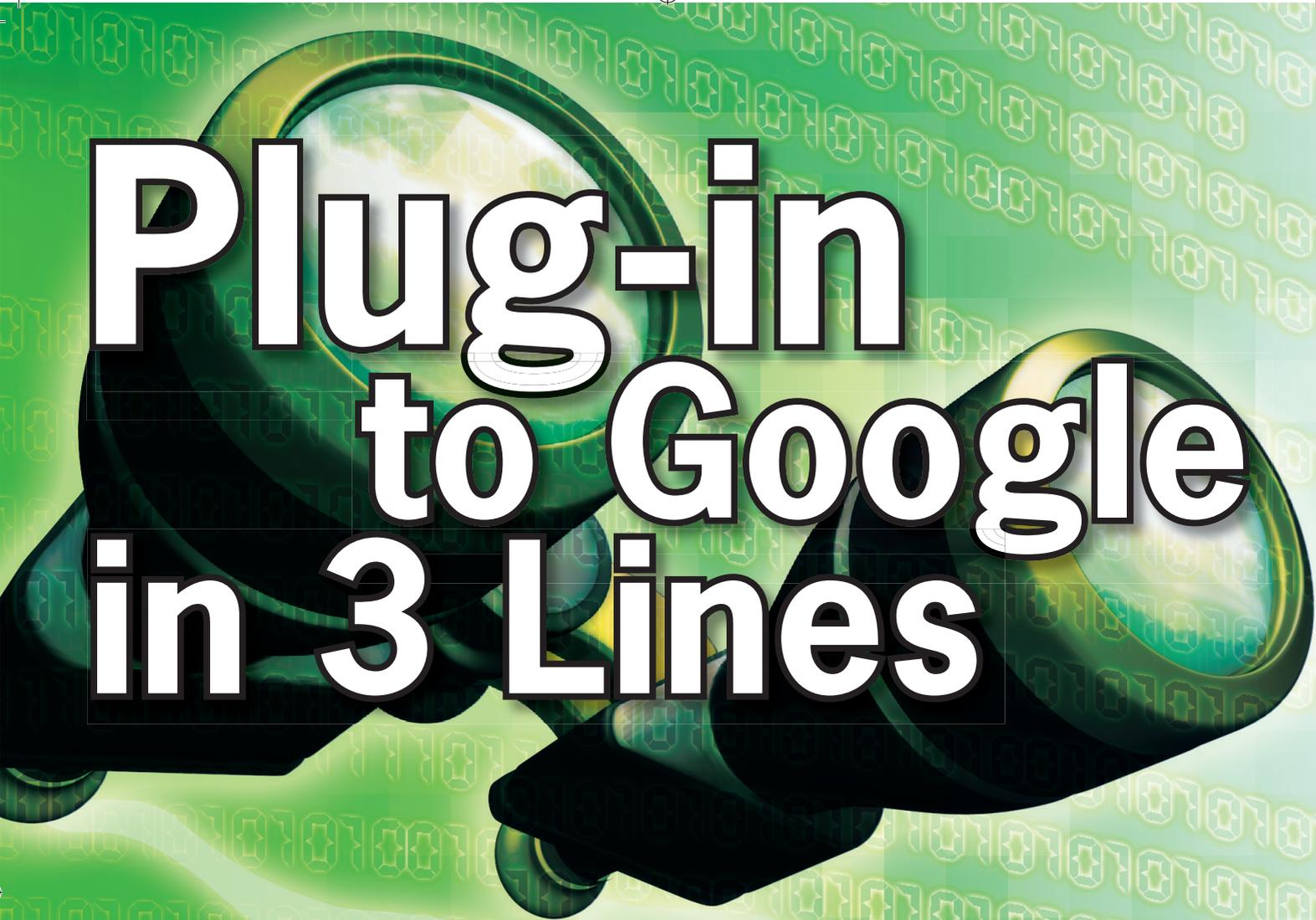


Figure 11. Modules of the Imagine image editor

Figure 12. Classes in the Paint API



Tim Boudreau (tboudreau@sun.com) had his first startup when he was 13, and has been hooked since, with brief departures to play rock and roll, write and play music and do graphics and photography. He is the coauthor of NetBeans, the Definitive Guide from O'Reilly and Associates, was part of the team that open sourced NetBeans and currently works as a developer on that project. Originally from Massachusetts, he lives in his adopted home city, Prague, in Czech Republic.



Plug-in to Google in 3 Lines

Integrating Google Code
Search in NetBeans

Geertjan Wielenga



The unofficial relationship between Google and NetBeans IDE started when Sun engineer Ludovic (“Ludo”) Champenois created on October 5, 2005, a Google Search Toolbar for NetBeans IDE. This Google integration was so popular, and Ludo’s implementation so cool, that it formed the basis of the very frequently used NetBeans Google Toolbar Module Tutorial. Since then, several enhancements have been built on top of this toolbar. For example, NetBeans user Ramón Ramos has added functionality allowing the user to select alternative search engines, or to provide (instead of a textfield) a right-click popup action on the selected line or word.

Now that the new Google Code Search has been announced (and to celebrate the one year anniversary of Ludo’s Google toolbar), it’s time for something else quite special. And here it is... **you can integrate Google Code Search in NetBeans IDE with only three lines of code.** That’s all that you need to do. The NetBeans APIs and NetBeans IDE module development functionality (wizards and tools in the Source Editor) do *everything else*.

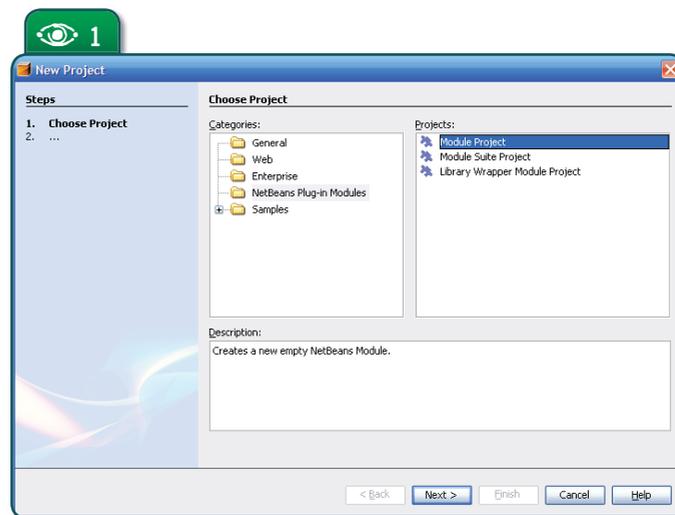
The NetBeans module

Follow along and you will be able to send a selection of code to Google Code Search, from a Java file open in the IDE’s Source Editor. In this article I’ve written the steps for the complete beginner, so if you’ve never written a NetBeans mod-

ule before, here’s your chance! This one could not possibly be simpler.

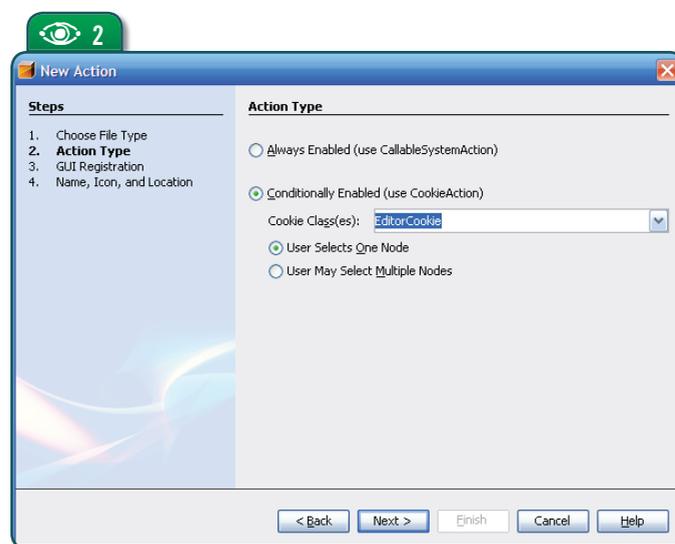
1. Creating the project

Choose *File/New Project*. In the New Project wizard, choose “Module Project” in the “NetBeans Plug-in Modules” category (see **Figure 1**). Click *Next*. Name the project “GoogleCodeSearch”, and browse to an appropriate location. Select “Standalone Module” and “Set as Main Project” (if these are not already selected by default). Click *Next* and then *Finish*.



 netbeans.org/download/dev/javadoc NetBeans APIs documentation

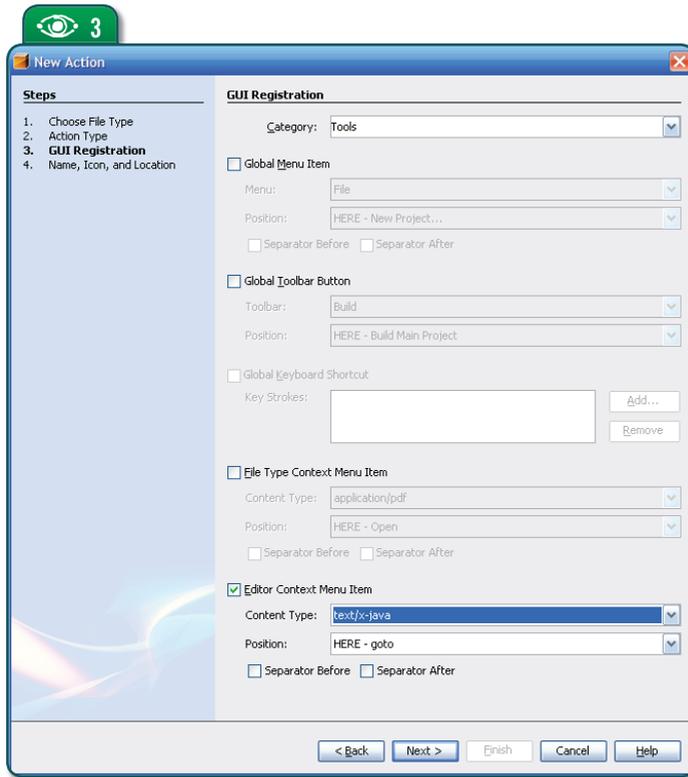
 **Figure 1.** Creating the Module Project



 **Figure 2.** Adding a conditional action

 google.com/codesearch Google Code Search

Figure 3.
Registering the action in the GUI



2. Adding and configuring an action

Right-click the project, choose *New*, and then choose *Action*. Select “Conditionally Enabled”, and in the *Cookie Classes* drop-down choose “EditorCookie” (see **Figure 2**). Click *Next*. In *Category* choose *Tools*, then unselect “Global Menu Item”. Select “Editor Context Menu Item”. Choose “text/x-java” from the *Content Type* drop-down list (this sets the MIME type, which determines the type of file your new action is applicable to) – see **Figure 3**. Click *Next*. Type “GoogleCodeSearch” in *Class Name*, and “Google Code Search” in *Display Name*. Click *Finish*.

3. The first two lines

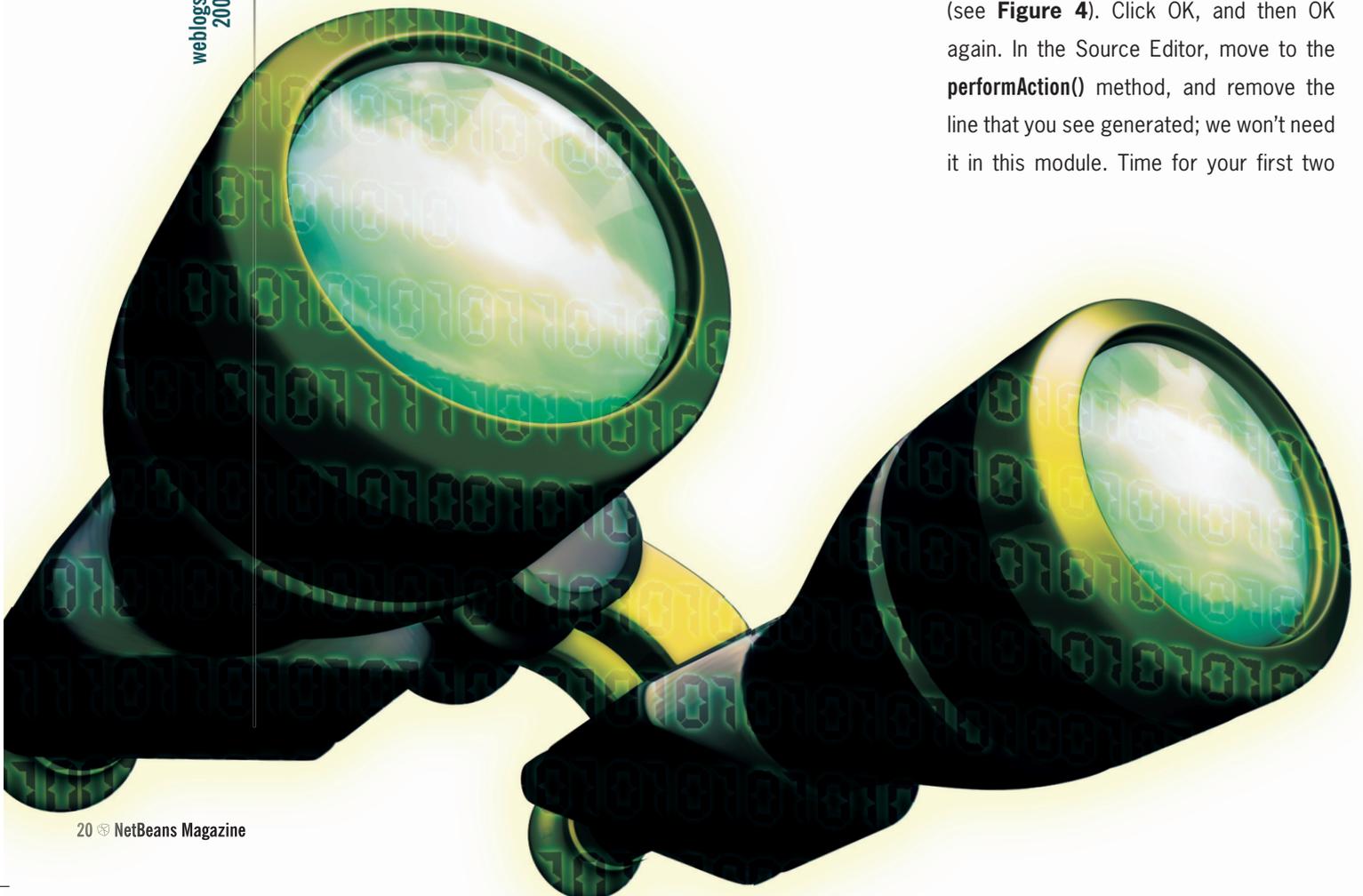
Right-click the project and choose *Properties*. Select the *Libraries* category, click *Add* and scroll to “Editor Library” (see **Figure 4**). Click *OK*, and then *OK* again. In the *Source Editor*, move to the `performAction()` method, and remove the line that you see generated; we won’t need it in this module. Time for your first two

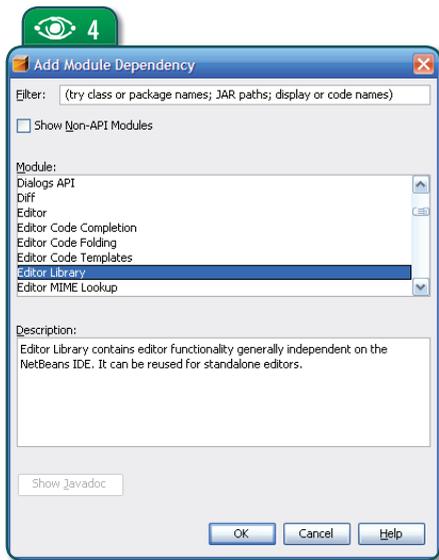
NetBeans
Google
Toolbar
Module
Tutorial

[platform.netbeans.org/
tutorials/nbm-google.html](http://platform.netbeans.org/tutorials/nbm-google.html)

Google
Search
Toolbar for
NetBeans
IDE

[weblogs.java.net/blog/ludo/archive/
2005/10/google_and_netb.html](http://weblogs.java.net/blog/ludo/archive/2005/10/google_and_netb.html)





lines of code! Add the following lines to `performAction()`:

```
JTextComponent editor=Registry.getMostActiveComponent();
String selection = editor.getSelectedText();
```

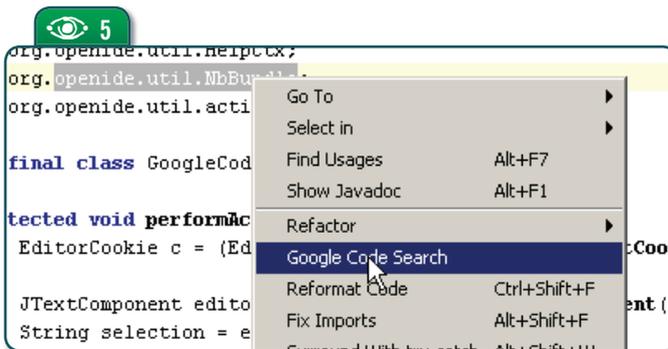
The first line becomes underlined in red, because you need **import** statements. Thanks to the dependency on “Editor Library”, you can click the lightbulb and choose `org.netbeans.editor.Registry`. Next, click on the lightbulb again and choose `javax.swing.text.JTextComponent`.

5. Completing the code

Now go back to the Project Properties dialog box, and add a dependency on “UI Utilities API”. Time for the final line of code! Here it is, at the end of `performAction()`:

```
URLDisplayer.getDefault().showURL(
    new URL("http://google.com/codesearch?hl=en&q="
        + selection + "&btnG=Google+Search");
```

When prompted for **import** statements,



you will need `java.net.URL` and `org.openide.awt.HtmlBrowser.URLDisplayer`.

A red underline will remain, because the `URL` constructor throws `MalformedURLException`. Click on the lightbulb and let the IDE surround the code with a **try-catch** block.

6. Installing and running

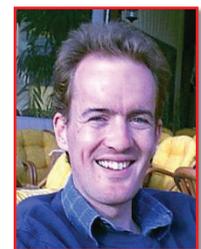
Hurray, you’re done! Right-click the module and choose “Install/Reload in Development IDE” (then click OK). NetBeans warns you that this action can be dangerous, but since this module is so simple, there’s little risk that we have a crasher bug here. So you can ignore the warning – the new module will be hot-deployed and activated in the same NetBeans instance you’re using. (The alternative would be loading a new NetBeans instance for testing.) Now select something in a Java file and choose the new menu item (see **Figure 5**).

Conclusions

Google Code Search is a powerful facility provided by Google. In this article you’ve seen how easy it is to integrate it into NetBeans IDE. Wizards did almost all the work, and you just needed to add three lines, the first two for selecting code and the last for sending the selection to your browser. However, although this module is simple, you have created a complete module that plugs into NetBeans and can be very useful in your development work. Congratulations, and have fun with NetBeans module development! ☺

Figure 4. Adding editing functionality

Figure 5. Selecting the menu item sends the selection to the IDE’s default browser, opening it in the Google Code Search page.



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





Java EE 5

in Action with

NetBeans

A critical exploration of Java EE 5's new productivity features, focusing on EJB 3.0, JPA, and NetBeans 5.5 tooling support

Oswaldo Pinali Doederlein

The traditional J2EE platform has long been regarded as powerful but difficult to learn and use: J2EE offers a massive number of high-end features and it's natural that these features have costs in complexity of architecture, APIs, and tools. Though NetBeans IDE has been offering strong support for J2EE development tasks for years, in an ideal world tools should not have to mask the inadequacies of frameworks. Frameworks should be well designed with respect to productivity – when this happens IDEs can be “unloaded” from the automation of dumb boilerplate code, and add an increasing number of higher-end productivity features without becoming, themselves, overloaded with complexity.

You will see in this article, which shows off both the new Java EE 5 functionality and NetBeans 5.5 features, that this is exactly the case for the next generation of enterprise Java development. We have a much easier, yet richer, framework, along with an increasingly powerful but still seamlessly integrated IDE, with effective support for the entire Java EE stack.

This article puts the major productivity enhancements of Java EE 5 through their paces. We'll look at annotation-driven APIs in general, EJB 3.0, the Java Persistence API (JPA), Dependency Injection, and application deployment. All this while going through a tutorial that develops a simple but complete application with NetBeans 5.5 and Glassfish/SJSAS, and looking at what NetBeans has to offer at the tool support side of Java EE 5. But differently from most tutorials that focus on teaching new APIs or techniques, we'll put some

time into analyzing the effectiveness of key Java EE features, extrapolating them to real-world projects.

Installing NetBeans and the Application Server

As this is NetBeans Magazine, I'll assume that you know how to download, install and configure the IDE and a compatible application server. At the time of writing, NetBeans offers support for SJSAS, GlassFish, JBoss, WebLogic, and JOnAS (with the JOnAS plugin), besides Tomcat. This tutorial was developed with the Sun Java System Application Server 9.0 (a.k.a. Java EE 5 SDK or Java EE 5 RI), but it's application-server-portable. Except that the example application uses SJSAS's predefined *jdbc/sample* data source – so in other servers, you may have to configure a database and data source.

The JPM Application

The example application allows you to keep track of all Java platforms, composed of JSR specifications and APIs grouped by packages – deserving the pompous name “Java Platform Manager”. More specifically, the application manages a simple database, and is implemented with the JPA, EJB and JSF APIs.

Granted, due to space limitations I provide only a tiny amount of the potential functionality of such an app. But when you reach the end, I hope you'll be so impressed by the ease of development of Java EE 5 with NetBeans 5.5, that you'll enjoy coding the rest of it.

Creating the Projects

Start by selecting *File | New Project > Enterprise/Enterprise Application*. In the *Name and location* tab, set *Project Name* to *JPM*, change the base directory to an appropriate location, and accept defaults for the other options (see **Figure 1**). This creates the projects *JPM*, *JPM-ejb* and *JPM-war*. NetBeans adopts a project structure that mirrors the Java EE deployment structure, with one project for each deployable module (respectively the EAR, EJB-jar, and WAR modules).

You'll notice that the deployment model is similar to the traditional J2EE's. But it was simplified, starting with the descriptor files, most of which are now optional or much simpler. For example, though NetBeans creates a *JPM-ejb/Configuration Files/MANIFEST.MF* and fills



Java EE 5 specification.

jcp.org/en/jsr/detail?id=244

its *Class-Path* directive, this is often unnecessary. A Java EE 5 container will add all JARs inside an EAR to the classpath of all modules of that EAR without a *Class-Path* directive. For this tutorial, you may remove the *MANIFEST.MF* file.

Configuration by Exception

In Java EE 5, you can omit most configuration items for which the container can reasonably guess a default value. This is clearly the case of *MANIFEST.MF/Class-Path*.

Explicit configurations are still supported, both for compatibility and for cases where defaults are not good enough. For instance, if you need to force a specific class lookup order among several JARs, or if you want each module to import only the subset of JARs it really needs, then *Class-Path* is still your friend. But these scenarios should be rather the exception than the rule.

Even the traditional *ejb-jar.xml* is now optional. Look inside *JPM-*ejb/Configuration Files**. You won't find any EJB descriptor, not even after creating EJBs. On the other hand, you may still need proprietary descriptors. For example, with SJSAS/Glassfish, the *JPM/Configuration Files/sun-application.xml* file is required for settings such as security role mappings. As it turns out, proprietary settings cannot be moved from descriptors to code annotations. This is technically possible, but not a good idea. Annotations live in the source code, and proprietary annotations would break portability.

Notice that the Configuration by Exception feature is not the same as the tool-supported defaults we're more familiar with. For example, even with J2EE 1.4, NetBeans won't require that you write the *ejb-jar.xml* descriptor by hand. It provides a visual editor, and the

wizards and editor provide default settings, e.g. *Transaction Type=Container* for all EJBs. But the container still requires the XML descriptor with all these settings written down explicitly, thus polluting your projects with additional files and redundant settings, and making its maintenance harder.

Additionally, Java EE defaults are often much smarter than simple fixed values for missing properties. The automatic generation of classpaths is one example of non-trivial default logic. A more sophisticated example is DDL generation for persistent entities.

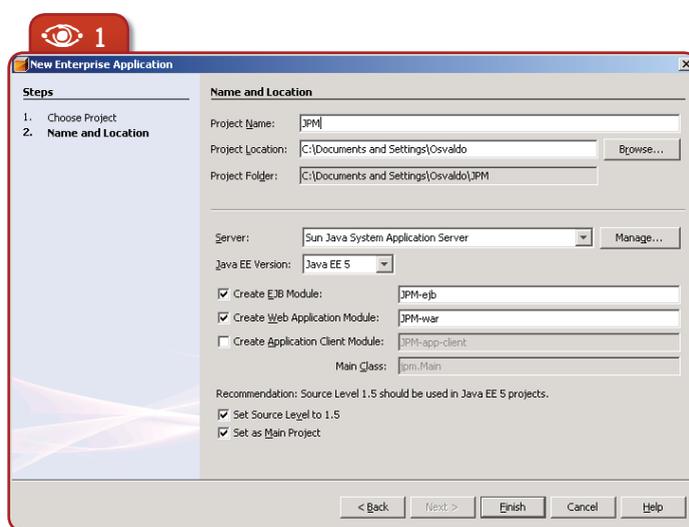
Implementing the Persistent Entities

Let's start coding the *JPM-*ejb** project, starting with the persistent entities. The application needs only three: Platform, JSR and Package. These will be persisted by the Java Persistence API (JPA).

The J2EE platform has been blamed for requiring a heavy load of bureaucratic artifacts even for simple tasks, so even a three-entity design like ours would be bloated with Entity Beans' home and bean interfaces, lifecycle methods and descriptors. Not to mention code that's not required (and not generated by IDEs) but is often needed in practice: a layer of POJOs to expose the Entity Beans to the outside world, DAOs to encapsulate the use of that code and convert between Entity Beans and POJOs, a Service Locator to provide resources like JNDI contexts, and so on.

This violates what I call **Complexity Scalability**. Simple problems should produce simple code, even if the framework supports much harder requirements like distributed transactions, clustering,

Figure 1.
Creating
the Java EE
projects.



declarative security and other high-end features of EJB. A good framework is “pay as you go”, not requiring any code or configuration that’s not really necessary. As a corollary, a *great* framework will allow even complex requirements to map to relatively simple code: the bookkeeping overhead, measured as a proportion of the “useful” lines of code, should scale linearly.

So, if a very simple artifact (like a persistent entity) can be programmed with 30 lines of code and 3 lines of overhead (like descriptor, annotation, external O/R mapping, implementation of API interfaces, or equivalent), then the framework’s overhead is 10% and its complexity scalability looks good – since a simple task is performed with low overhead. For a much more complex artifact, like a thousand-line persistent class with many relationships, if the overhead is kept at 10% (100 lines), the complexity scalability is good; if it drops proportionally, e.g. to 3% (30 lines), it’s awesome.

It’s an interesting exercise to evaluate APIs and frameworks by this criterion. Take Hibernate, for example. The framework is regarded as easy to program, but its base overhead is high. The mapping files for “dumb” persistent classes (with no business logic) are often more complex than the classes themselves, because there’s no default logic at all. Every property and relationship, however trivial, must be explicitly mapped, and the mapping is highly redundant with the code. The same holds for Spring applications. These often contain so much XML metadata that one wonders how this is any easier than keeping the same information in Java code. (Note that I’m looking only at ease of program-

ming, not flexibility or other factors.)

In traditional J2EE, of course, we have the worst of the two worlds: an Entity Bean requires *both* a lot of code *and* a great deal of metadata. And its overhead scales linearly at best: complex apps require increased amounts of settings – for example, lots of resource references from beans to other beans, data sources, queues etc. – and the framework does nothing to avoid the need to declare settings explicitly. Thus complex apps easily end up with numerous, long descriptors.

The Persistence Unit

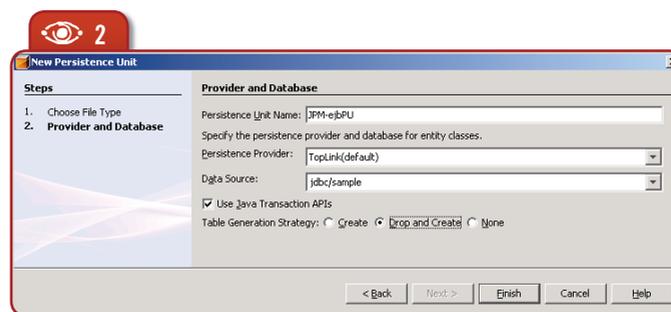
Before we can create any persistent entity, the JPA requires definition of a Persistence Unit. A P.U. is a set of JPA configurations (in other words, a descriptor). From the *JPM-ejb* project, run *New>File/Folder>Persistence>Persistence Unit* (**Figure 2**).

The only interesting option here is *Persistence Provider*: for SJSAS or GlassFish, select *TopLink(default)*. TopLink is an O/R mapping product from Oracle, but its Essentials version is open source. Users of other application servers may have to select a different provider. NetBeans comes preconfigured also for Hibernate and KODO.

 You can also ignore your application server’s default JPA provider and install a different one like Hibernate in SJSAS. This is a big advantage of Java EE 5: JPA supports “pluggable” persistence providers, instead of forcing you to go with each server’s built-in implementation like in EJB/CMP. The loosely-coupled design of JPA goes even further: you can use it in Java SE (without an application server), and you can make the Persistent Unit descriptor refer to external proprietary descriptors, like Hibernate’s *.hbm* files. This helps immensely in migrating legacy code. NetBeans supports this flexibility by offering multiple providers, and allowing you to configure new providers if necessary.

The resulting *Configuration Files/persistence.xml* file is a minimalist descriptor, only mandatory because it’s the only way to specify “global” properties like the persistence provider.

You could fill *persistence.xml* with the names of all persistent



 **Figure 2.** Creating a Persistence Unit.

classes, but this is not normally needed, as the default behavior is to automatically find all such classes (which is always good enough if your EJB module contains a single P.U.). Finally, the *persistence.xml* could include detailed mapping information, but this is also optional because we can do all the mapping with code annotations (or even more easily with default JPA behaviors).

The Platform entity

From the *JPM-ejb* project, select *New>Entity Class*. This brings up the dialog shown in **Figure 3**. Make *Class Name* = **Platform**, *Package* = **jpm.entity** and *Primary Key Type* = **String**. This wizard will generate the code in **Listing 1**. This is simply a POJO, but let's consider some important facts:

- The class uses some JPA annotations: **@Entity** makes it a persistent entity, **@Id** selects the attribute used for the primary key (PK), and **@GeneratedValue** declares that the value for this PK should be generated automatically.
- The entity must offer a default constructor, and JavaBeans-style getters and setters for its persistent attributes or relationships.

Only the code in boldface is mandatory. The rest – **hashCode()**, **equals()**, **toString()**, **@Override** annotations, implementation of **Serializable** – is just bonus from NetBeans. They are good programming practices, but not required by the JPA. We'll omit most of this non-required code to simplify future listings. The exception is **implements Serializable**: this is not required by the JPA but is necessary in our application because we want to transfer the entity objects through (possibly remote) EJB calls.

The entities will double as DTOs (data transfer objects)¹. In fact, one of the first advantages noticed when moving from a traditional ORM tool to a POJO-based one is that there's no need to write redundant POJOs / Value Objects / DTOs. You also don't need any code to convert between persistent and non-persistent objects, nor the structure (like the DAO pattern) to keep all that code organized.

 JPA only requires two kinds of classes to be serializable: classes that map composite PKs (with two or more columns), and the classes of attributes you may want to map to a LOB column. Anyway, you're better off if you follow the best practice that all POJOs should be serializable.

The *New Entity* wizard always generates the **@GeneratedValue** annotation, but our application uses only natural keys, so you must also

delete this annotation from the **id** attribute. In fact, **@GeneratedValue** is illegal here because only numeric PKs can be auto-generated.

 The entity's **hashCode()** and **equals()** methods are not necessary because the JPA will rely only on the PK for identification and hashing of entities. But if you use a custom class as primary key (which is required to support multi-column keys), your PK class must provide correct implementations of **hashCode()** and **equals()**.

JSR and Package entities

Using the same *New Entity* wizard, create two new classes: **JSR** (*Primary Key Type* = *int*) and **Package** (*Primary key type* = *String*). In the generated code, start by deleting the **@GeneratedValue** annotations. Next, rename the **id** attributes of these two classes to **JSR.number** and **Package.name**, respectively. Rename the getter and setters accordingly too.

You will notice that NetBeans balks at the new attribute names, showing a warning marker in the source editor with the hover message "The column name of the persistent field is a SQL-99 keyword". Yes, too bad – **number** and **name** are reserved SQL keywords, and this can cause problems in the generated tables because the JPA engine will map each attribute to a column with the same name, by default. Fix this by adding **@Column** annotations, e.g. **@Column(name="package_name")** to the **name** attribute, which will remove the warnings.

Relationships

Our **Platform** entity has a many-to-many relationship with the **JSR** entity (for example, JSR-224, the JAX-WS API, is part of both

¹ Remember, the term DTO implies a *role*: objects that can be transferred through a specific middleware; whereas POJO implies *structure*: objects whose implementation does not depend on specific APIs.

Java EE 5 and Java SE 6). Likewise, **JSR** has a one-to-many by-value aggregation relationship with **Package**. Both relationships are bidirectional.

Start declaring a `List<JSR> jsrs` attribute in **Platform**. Again NetBeans will show a warning: “The multi-valued entity relation is not defined”. If you left-click the attribute name or select the warning marker and right-click, NetBeans will offer two corrections for the problem. Select *Create bidirectional “ManyToOne” relationship*. This shows the wizard in **Figure 4**.

A bidirectional relationship requires that the entity in the other end, **JSR**, point back to **Platform**. Accept the *Create new field* option and `Name = platforms`. You must add the getters and setters manually for

the new **Platform.jsrs** and **JSR.platforms** relationships. It’s useful also to have a constructor that receives only the PK and another receiving all attributes. The result should be close to **Listing 2**, where are highlighted the parts introduced after

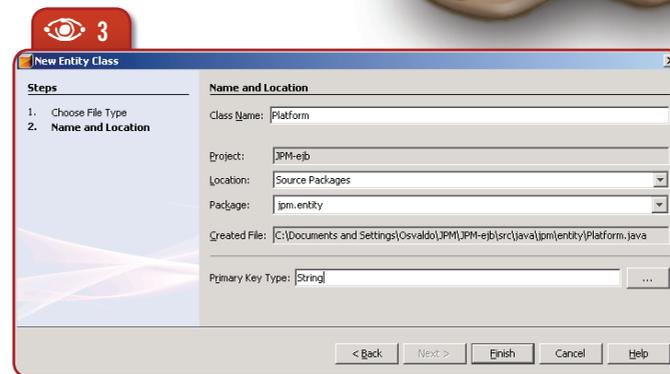


Figure 3. Creating an Entity Class.

Listing 1. Original code emitted for the Platform entity class (generated comments omitted).

```
package jpm.entity;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Platform implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private String id;
    public Platform() {
    }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.id != null ? this.id.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (!(object instanceof Platform)) {
            return false;
        }
        Platform other = (Platform)object;
        if (this.id != other.id && (this.id == null || !this.id.equals(other.id))) return false;
        return true;
    }
    @Override
    public String toString() {
        return "jpm.entity.Platform[id=" + id + "]";
    }
}
```

 **Figure 4.**
Creating a
relationship.

creating the original entity code skeleton.

Note that the relationship wizard annotates **Platform.jsrs** with **@ManyToMany**, and the new **JSR.platforms** with **@ManyToMany(mappedBy="jsrs")**. This **mappedBy** property is intended to avoid ambiguities, should **Platform** contain multiple collections whose element-type is **JSR**. In our code this is not the case, so you could get rid of that **mappedBy**, unless you think it makes the code easier to read (in this case, you could add **mappedBy="platforms"** to **Platform.jsrs**'s annotation).

Now you can finish the **JSR** class by creating another bidirectional relationship with **Package**. This relationship will have a **OneToMany** cardinality, so **JSR** will have an attribute of type **List<Package>** but **Package** will only need a simple reference to a **JSR**. **Listings 3** and **4** show the final code for these classes. I added a couple of tweaks to the relationship annotations (which will be explained later).

 **Listing 2.** The complete Platform class.

```
package jpm.entity;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.persistence.*;
import static javax.persistence.FetchType.*;
import static javax.persistence.CascadeType.*;

@Entity
public class Platform implements Serializable
{
    @Id
    private String id;
    @ManyToMany(fetch=EAGER, cascade={
        MERGE, PERSIST, REFRESH})
    private List<JSR> jsrs;
    public Platform () {
    }
    public Platform (String id) {
        setId(id);
    }
    public Platform (String id, JSR... jsrs) {
        this(id);
        setJsrs(new ArrayList<JSR>(Arrays.asList(jsrs)));
    }
    public String getId () {
        return this.id;
    }
    public void setId (String id) {
        this.id = id;
    }
    public List<JSR> getJsrs () {
        return jsrs;
    }
    public void setJsrs (List<JSR> jsrs) {
        this.jsrs = jsrs;
    }
}
```



Testing the Deployment

The application's persistent entities are complete, so let's test them. In the **JPM** project run *Deploy project*. In a few seconds you should see messages reporting success in the console windows.

The deployment operation will also create the necessary tables, because we have set the Persistence Unit's *Table Generation Strategy* to *Drop and Create* (or *Create*). You can check this by connecting to the project's data source. If you're using the recommended JavaDB sample database in **SJSAS**, this will be predefined in *Runtime>Databases>jdbc:derby://localhost:1527/sample [app on APP]*. Just run this node's *Connect* command, log in with the password *app*, and you'll see the created tables. Notice, in particular, that an associative table, **PLATFORM_JSRS**, is created to map the **ManyToMany** relationship between **Platform** and **JSR**.

Annotations: Intelligence and Limitations

As this article intends also to make a critical review of Java EE 5's ease-of-use features, we can't stop here, implying that all persistence-related work will always be as easy as in our simple example.

Many developers starting to use annotation-enabled APIs think that the new APIs just shuffle things around – metadata that used to be written in XML descriptors is

now written in code annotations but they have to be written just like before, right?

Not really. As you can see in our example, we *don't* have to create annotations for all metadata previously in descriptors. For example, there are no annotations declaring that **JSR.description** is a persistent property, nor declaring the mapped table for each entity. And we only needed two **@Column** annotations for attributes whose names would map to illegal SQL identifiers. We could have been more verbose, writing code like:

```
@Entity
@Table(name="T_PLATFORM", schema="JPM")
public class Platform implements Serializable {
    ...
    @Column(name="PK_ID", nullable=false,
            length=512).
    private String id;
    ...
}
```

But these annotations are all optional. If they are missing, the JPA will look at the names and types of classes and attributes, so a class named **Platform** becomes a table named **PLATFORM**; a **String** attribute becomes a **VARCHAR** column, and so on. (If you want an entity class to contain an attribute *not* mapped to any column, tag it with the **@Transient** annotation).

 In principle, a framework based in descriptors, like EJB 2.1 or Hibernate, could also infer default configurations from the application classes. But the tight coupling of code and annotations, plus the availability of standard and built-in APIs and tools to manipulate annotated classes, make this strategy more robust, efficient and easy to implement.

Listing 3. The JSR class.

```
package jpm.entity;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import javax.persistence.*;
import static javax.persistence.FetchType.*;
import static javax.persistence.CascadeType.*;

@Entity
public class JSR implements Serializable
{
    @Id
    @Column(name="num")
    private int number;
    private String description;
    @ManyToMany(mappedBy="jsrs")
    private List<Platform> platforms;
    @OneToMany(fetch=EAGER, cascade=ALL)
    private List<Package> packages;
    public JSR () {
    }
    public JSR (int id) {
        setNumber(id);
    }
    public JSR (
        int id, String description, Package... packages) {
        this(id);
        setDescription(description);
        setPackages(new ArrayList<Package>(
            Arrays.asList(packages)));
    }
    public int getNumber () {
        return this.number;
    }
    public void setNumber (int number) {
        this.number = number;
    }
    public String getDescription () {
        return description;
    }
    public void setDescription (String description) {
        this.description = description;
    }
    public List<Platform> getPlatforms () {
        return platforms;
    }
    public void setPlatforms (List<Platform> platforms) {
        this.platforms = platforms;
    }
    public List<Package> getPackages () {
        return packages;
    }
    public void setPackages (List<Package> packages) {
        this.packages = packages;
        if (packages != null) for (
            Package p: packages) p.setJsrs(this);
    }
}
```

Our sample application is also simplified by completely automatic DDL generation. Even if you have to write some annotations detailing types, table and column names, nullability and other properties, you

Listing 4. The Package class.

```

package jpm.entity;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Package implements Serializable
{
    @Id
    @Column(name="package_name")
    private String name;
    @ManyToOne
    private JSR jsr;
    public Package () {
    }
    public Package (String name) {
        setName(name);
    }
    public String getName () {
        return this.name;
    }
    public void setName (String name) {
        this.name = name;
    }
    public JSR getJsr () {
        return jsr;
    }
    public void setJsr (JSR jsr) {
        this.jsr = jsr;
    }
}

```

never have to collect these pieces into a **CREATE TABLE** statement. Thanks to automatic schema generation and SJSAS's embedded JavaDB database, we could write the whole app without even knowing that a relational database is being used! These features allow you to go far in your coding before you start worrying about "real-world" issues.

From Hello World to the Real World

Now, how realistic is the persistence code in this tutorial? In real applications you often have to comply with strict rules for database schemas; you may have to use legacy tables, or follow a style guideline that says something like "all tables should have the prefix 'T_'. In the average case, you'll be forced to tag every entity class with a **@Table** annotation, and every property with **@Column** or related annotations (**@JoinColumn** and **@JoinedColumns**, not used in this article). In the worst case, you may have to give up on the DDL generation feature, for example because your **CREATE TABLE** statement must be highly tuned with fine-grained (and usually proprietary) clauses, like physical allocation or partitioning options.

So in the worst case, our entity classes may be relatively polluted

by many annotations, or else we'll have to write DDLs and manually keep them in sync with the classes. But this doesn't mean the JPA's default behaviors are useless, but only that you pay as you go. The JPA (and other annotation-enabled APIs in Java EE 5) requires a programming effort that's proportional to the complexity of the problem being solved, and not artificially inflated by bookkeeping artifacts that must be written whether you need them or not.

Implementing the Session Bean

Now that we're done with persistence, the next step is to create a Session Bean that exposes methods to manipulate and query the persistent entities.

From the *JPM-ejb* project, run *New>Session Bean*. Make *EJBname=JavaPlatformManager* and *Package=business*. Accept defaults for *Session Type=Stateless* and *Create Interface=Local*, but check *Create Interface=Remote*. The wizard will create the skeletal sources for the Session Bean. These include the class **JavaPlatformManagerBean**, and the interfaces **JavaPlatformManagerLocal** and **JavaPlatformManagerRemote**. You'll just need to fill the blanks to obtain the code in **Listings 5** and **6**.

Listing 5 shows the local interface for the **JavaPlatformManager** bean. We have a **@Local** annotation, but we don't need to extend the interface **EJBLocalObject** anymore. So we have a POJI (*Plain Old Java Interface*).

The **JavaPlatformManagerRemote** interface (not listed) is identical, except that **@Local** is replaced by **@Remote**. When you have both local and remote interfaces

^{1,2} Using a single Java interface and annotating it with both **@Local** and **@Remote** could be even easier, but EJB 3.0 does not support this further simplification.

for a Session Bean, EJB 3.0 allows you to refactor all common methods to another interface, which is extended by both the Local and the Remote interfaces². This is a big improvement from EJB 2.x, where this refactoring and type unification was impossible because the remote interface was required to declare all methods with **throws RemoteException** (EJB 2.0: 7.10.5) but the local interface shouldn't use this exception (EJB 2.0: 7.10.7, 18.3.8, 18.6).

Listing 6 is the full implementation of our Stateless Session Bean, qualified as such by the **@Stateless** annotation. Notice how this class implements both the local and remote interfaces. In EJB 3.0, we finally have the complete strong-typing that EJB has lacked since its first version.

Now let's analyze the bean implementation. The **@PersistenceContext** annotation does dependency injection for the **EntityManager**, a JPA object that stands behind a Persistence Unit. When the **JavaPlatformManagerBean** class is instantiated by the container, the **EntityManager em** attribute will be initialized before any method can be invoked.

The method **listPlatforms()** runs a query that returns all **Platform** objects that were found in the database. These objects will have their **jsrs** collections populated, because we added the option **fetch = EAGER** to the relationship's annotation. The **EAGER** op-



Listing 5. The Session Bean's Local interface.

```
package jpm.session;

import java.util.List;
import javax.ejb.Local;
import jpm.entity.Platform;

@Local
public interface JavaPlatformManagerLocal
    extends JavaPlatformManager {
    void createPlatform (Platform plat);
    List<Platform> listPlatforms ();
}
```

Listing 6. The Session Bean's implementation.

```
package jpm.session;

import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.*;
import jpm.entity.Platform;

@Stateless
public class JavaPlatformManagerBean
    implements jpm.session.JavaPlatformManagerRemote, jpm.session.
        JavaPlatformManagerLocal {
    {
    @PersistenceContext private EntityManager em;
    public void createPlatform (Platform plat) {
        em.merge(plat);
    }
    public List<Platform> listPlatforms () {
        return em.createQuery(
            "select p from Platform p").getResultList();
    }
}
```



Java EE 5 SDK/
SJSAS, the Java
EE 5 Reference
Implementation.

tion in our example, forces a **Platform** entity's associated **JSRs** to be read when the **Platform** is read (possibly through a single query with an outer join, for better performance – a critical optimization if you need the associated objects most of the time you need the parent object). Likewise, the **JSR.packages** collections will also come initialized.

JPA versus EJB

The **fetch = EAGER** option is also important for correctness in the example application, because we return the **Platform** objects through a (possibly remote) EJB invocation. Entity instances associated with an **EntityManager** are managed by the JPA. If we used **fetch = LAZY** (the default), the query could return **Platform** objects with uninitialized **jsrs** collections. These would be populated on demand, upon first invocation of **getJsrs()**, but this only works for the original, JPA-managed collections.

When you transfer these objects through an EJB call, the application at the other side of the call receives different objects (reconstructed through cloning or serialization/deserialization). The new objects are not bound to an **EntityManager**. So when the receiver invokes **getJsrs()**, the JPA engine won't do its magic of intercepting the call and fetching the collection on demand. In fact, the receiver will often be an application – like our *JPM-war* – that doesn't contain a persistence unit, and hasn't even access to the entities' data source.

This issue is well-known to users of other POJO-based ORMs like Hibernate, and the solution is the same: eager fetching. (And just like in Hibernate, instead of tagging the relationship with a **fetch** option that affects all loads, you can write queries with a **FETCH JOIN** clause that gives finer control over when this loading strategy should be used.)

Who moved my lifecycle?

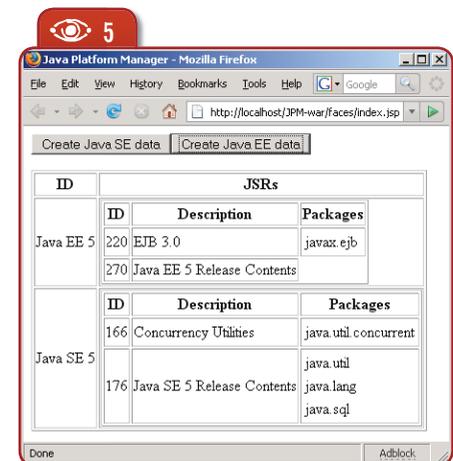
Just like for JPA, the “pay as you go” rule also holds for programming EJB 3.0 components. For example, our session bean didn't need to implement any of the traditional lifecycle methods, such as **ejbActivate()**. But that does not mean that lifecycle control is gone from Java EE. You can still hook into events like creation or activation, but these are now specified by annotations rather than interfaces. For example:

```
@PostActivate
public void postActivate (InvocationContext ctx)
{
    // Actions to execute after the activation
}
```

Annotations are finer-grained than interfaces like **SessionBean**: you can implement only one of the lifecycle methods, ignoring all others.

Differently from the JPA entities, however, when programming EJBs you'll find that most optional methods and annotations are rarely necessary, even in large, real-world applications. In my experience with good old J2EE, I rarely had to implement any of these lifecycle methods (well, if you don't count paranoid-logging implementations that just report “*Bean XYZ was Activated*”).

 **Figure 5.**
The Web GUI.



So the EJB 3.0 programming model results in much tighter code, even for realistic applications.

Implementing the Web client

Our application's back-end is ready. All that's missing is a GUI. I'm not much of a GUI programmer, but life has taught me that most people won't believe my code does

anything useful otherwise, so I'll provide a simple web GUI with the subproject *JPM-war*.

Start by configuring the NetBeans project to enable JSF. In the *JPM-war* project's properties page, select *Frameworks>Add*, and pick *JavaServer Faces*. This will populate the project with the minimum required configurations (including sample JSPs we won't use; delete those).

The Web GUI contains a single JSP that allows



Listing 7. The Web GUI's view implementation, *index.jsp*.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<html>
<head>
  <title>Java Platform Manager</title>
</head>
<body>
  <f:view>
    <h:form id="CreateTestData">
      <h:commandButton value="Create Java SE data" action="#{JPMClient.createJavaSE}"/>
      <h:commandButton value="Create Java EE data" action="#{JPMClient.createJavaEE}"/>
    </h:form>
    <h:dataTable value="#{JPMClient.allPlatforms}" var="platform" border="1">
      <h:column>
        <f:facet name="header">
          <h:outputText value="ID"/>
        </f:facet>
        <h:outputText value="#{platform.id}"/>
      </h:column>
      <h:column>
        <f:facet name="header">
          <h:outputText value="JSRs"/>
        </f:facet>
        <h:dataTable value="#{platform.jsrs}" var="jsr" border="1">
          <h:column>
            <f:facet name="header">
              <h:outputText value="ID"/>
            </f:facet>
            <h:outputText value="#{jsr.number}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Description"/>
            </f:facet>
            <h:outputText value="#{jsr.description}"/>
          </h:column>
          <h:column>
            <f:facet name="header">
              <h:outputText value="Packages"/>
            </f:facet>
            <h:dataTable value="#{jsr.packages}" var="pack" border="0">
              <h:column>
                <h:outputText value="#{pack.name}"/>
              </h:column>
            </h:dataTable>
          </h:column>
        </h:dataTable>
      </h:column>
    </h:dataTable>
  </f:view>
</body>
</html>
```

glassfish.dev.java.net
Project
GlassFish.

you to create sample **Platform** objects, and shows the ones already present in the database. **Listing 7** shows the source for this JSP, which is built with JSF component taglibs. **Figure 5** shows the page in action.

The JSP requires a Managed Bean – a POJO that is registered in the `WEB-INF/faces-config.xml` descriptor, so JSF-based pages can access the bean's properties and invoke their methods. Create it with `New>File/Folder>Web>JSF Managed Bean`; set `Class Name` to **JPMClient** and `Package` to **jpm.web**, accepting other defaults. This bean will provide the following services to our JSP:

- The property **allPlatforms** returns all **Platforms** (including child **JSR** objects and **Packages**), that the page will dump as nested HTML tables with JSF's `<h:dataTable>` components.
- The methods **createJavaSE()** and **createJavaEE()** are event handlers for JSF's `<h:commandButton>` component's **action** events. Each of these buttons triggers the insertion of one sample **Platform** object in the database (after which the page will be updated).

Listing 8 shows a simple controller implementation. There's no need of JNDI code to look up the Session Bean: just annotate it with `@EJB`, and dependency injection will do the rest. You don't even have to handle **RemoteException**, so the invocations to this EJB are

straightforward even though we're using a remote interface.

Looking again at **Listing 6**, check out the implementation of the EJB's **createPlatform()** method. You only have to pass the **Platform** object to **EntityManager.persist()**. The child **JSR** and **Package** objects are persisted by reachability. This happens because we declared the relationship **Platform.jsrs** with the option **cascade = {MERGE, PERSIST, REFRESH}**, and **JSR.packages** with **cascade = ALL**.

The **cascade** rule indicates which persistence operations are propagated by reachability to associated objects. For **Platform.jsrs** we used **MERGE** (synchronization of the object in memory with persistent records), **PERSIST** (creation) and **REFRESH** (reloading of persistent state to the object in memory). For **JSR.packages** we used **ALL**, which combines all the former with **DELETE** (dele-

 **Listing 8.** The Managed Bean.

```
package jpm.web;

import java.util.*;
import javax.ejb.EJB;
import jpm.entity.JSR;
import jpm.entity.Platform;
import jpm.entity.Package;
import jpm.session.JavaPlatformManagerRemote;

public class JPMClient {
    @EJB private JavaPlatformManagerRemote jpm;
    public void createJavaSE () {
        jpm.createPlatform(new Platform("Java SE 5",
            new JSR(176, "Java SE 5 Release Contents",
                new Package("java.lang"), new Package("java.util"), new Package("java.sql")),
            new JSR(166, "Concurrency Utilities", new Package("java.util.concurrent"))));
    }
    public void createJavaEE () {
        jpm.createPlatform(new Platform("Java EE 5",
            new JSR(270, "Java EE 5 Release Contents"),
            new JSR(220, "EJB 3.0", new Package("javax.ejb"))));
    }
    public List<Platform> getAllPlatforms () {
        return jpm.listPlatforms();
    }
}
```

tion of the persistent data).

The default is not to cascade any operation. You should of course use the most appropriate option for each kind of relationship. Here, **Platform**↔**JSR** is a simple association, so you could remove a **JSR** from a **Platform** without destroying the **JSR**, which is an independent entity (there are some **JSRs** that are part of multiple **Platforms**, and others that aren't part of any). But **JSR**↔**Package** is an aggregation by value. If you delete a **JSR**, its **Packages** should be deleted too.

Testing the complete application

Redeploy the application and load the following URL in your browser: `http://localhost/JPM-war/faces/index.jsp`. (The URL will be different if you've installed the application server to another HTTP port than 80 or used a different name for the *JPM-war* project). Click both buttons; after each click a new **Platform** will be inserted and displayed in the updated page. Don't click any button twice, as this would cause a PK violation in the database.

Our job is not done, however, before running the *Verify project* command on all projects. This submits each project to a massive number of Java EE-specific validations (in the results window I recommend selecting *Display = Failures and Warnings only*). Make sure you're clean on these validations to ensure both correctness and portability.

Look ma, no JNDI!

You may have noticed that we not only avoided the JNDI APIs, but also didn't even

have to specify JNDI names. This is because EJB 3.0 generates default JNDI names for all beans and even for resource references that are implicit in dependency injection annotations like **@EJB**.

You can ignore these defaults and force explicit JNDI names; for example, **@Stateless(name="ejb/session/JPM")** and **@EJB(name="ejb/session/JPM")**. But once again EJB 3.0 default behavior here is widely applicable. I've seen few EJB applications in which JNDI names are complex enough as not to allow use of Java EE 5 defaults. For example, for the *JPMClient.jsp* resource reference, the system will generate the default JNDI name `java:comp/env/jpm.web.JPMClient/jpm`, which, by combining the fully-qualified class name and field name, should be unique enough not to cause clashes.

Conclusions

Java EE 5 is a major ease-of-programming release, built around many novel design ideas: configuration by exception, POJO-oriented programming, annotation-driven APIs, dependency injection, a state-of-the-art persistency API, and a streamlined component model without hard-line rules like throwing **RemoteException**.

Nevertheless, Java EE is still a large, complex framework with a boatload of features – and good IDE support still makes a big difference. NetBeans 5.5 offers a full range of Java EE-specific support: code-generation wizards; visual descriptor editors; code completion, validation and automatic fixes for annotations; tight integration with application servers and building of Java EE deployment packages, as well as JSP debugging, support for both Struts and JSF, and many useful utilities like database browsing or HTTP monitoring. And we didn't even get started with the *really* advanced features, like SOA/BPEL tools in the Enterprise Pack. All these features come out-of-the-box, without need of dealing with third-party plug-ins.

The recent advances in the Java EE platform and the NetBeans IDE should not only make enterprise-grade development simpler, but also easier to learn and more incremental. Many tedious and error-prone tasks of the past are now gone, have been rationalized, or are highly automated by powerful IDE support. This translates directly to higher productivity for all developers and a gentler learning curve for beginners. ☺



Osvaldo Pinali Doederlein

(opinali@gmail.com)

is a software engineer and consultant, working with Java since 1.0beta. An independent expert for the JCP, he has served for JSR-175 (Java SE 5), and is the Technology Architect for Visionaire Informatica, holding a MSc in Object Oriented Software Engineering. Osvaldo is a contributing editor for Java Magazine and blogs at weblogs.java.net/blog/opinali.



JAVA SE WEB SERVICES

**Create, consume and
deploy basic web
services productively
using NetBeans**

Milan Kuchtiak

This article illustrates how a web service can be created and tested as a NetBeans standard Java Project. The new NetBeans 5.5 integrated support for JAX-WS 2.0 enables you to easily create and consume web services.

Creating the web service

Start NetBeans 5.5, then create a new Java Application project and name it "GeometricalWS". In the project's *Properties* dialog select the *Libraries* category and add "JAX-WS 2.0" (see **Figure 1**). This step

is necessary only if you're using Java SE 5 or lower. Java SE 6 already includes the JAX-WS APIs.

Now create a Java class named "CircleFunctions" with the code shown in **Listing 1**. The `@WebService` annotation makes the class a web service. The other annotations declare the web service's operations and their parameters, influencing the automatic generation of a WSDL document for this class.

Using the `javax.xml.ws.Endpoint.publish()` method, the web service can be deployed to a simple web server provided by the JAX-WS runtime. Update the project's **Main**

class with the code from **Listing 2**. Notice that the `publish()` method requires the URL address for the web service and an instance of the **CircleFunction** class. The latter will be invoked to serve requests.

Run the application. The message in the output window will notify you that the web service was published successfully:

```
Web service was published successfully.
WSDL URL: http://localhost:8765/GeometricalWS/CircleFunctions?WSDL
```

To check that the web service was really published, launch your web browser and open the web service URL: `http://localhost:8765/GeometricalWS/CircleFunctions?WSDL`. The browser should show a WSDL file.

Steps to create a Client

Developing a web service client with NetBeans 5.5 is even simpler.

Listing 1. The CircleFunctions class, a full web service implementation.

```
package geometricalws;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;

@WebService(name="Circle", serviceName="CircleService", portName="CirclePort")
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class CircleFunctions {

    @WebMethod(operationName="area")
    public double getArea(@WebParam(name="r") double r) {
        return Math.PI * (r * r);
    }

    @WebMethod(operationName="circumference")
    public double getCircumference(@WebParam(name="r") double r) {
        return 2 * Math.PI * r;
    }
}
```

blogs.sun.com/pblaha/entry/developing_web_services_for_mustang

Petr Blaha's blog: Developing Web services for Mustang in NetBeans.

java.sun.com/javase/downloads/index.jsp

Java EE 5 SDK/SJSAS, Java EE 5's Reference Implementation.

1

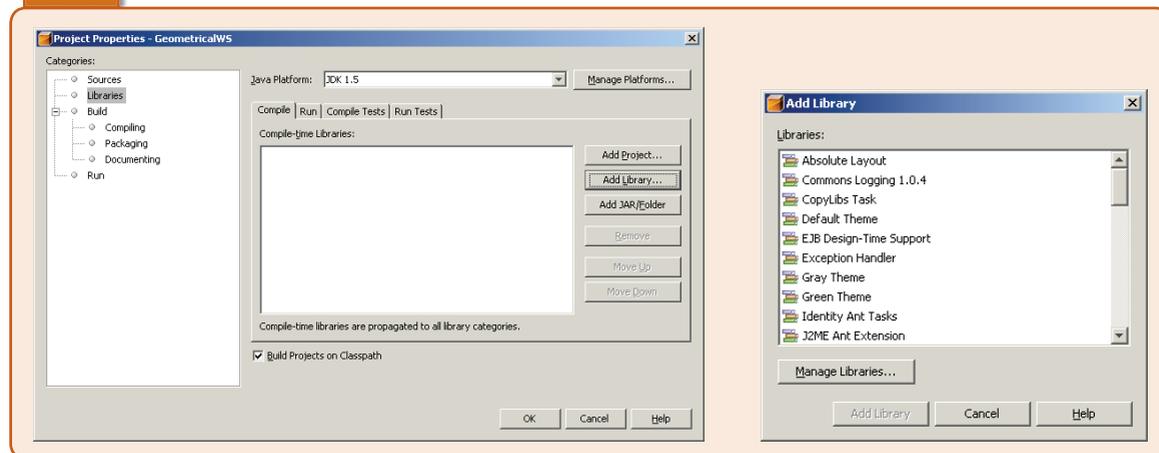


Figure 1. Adding JAX-WS support to a common Java project (only necessary with Java SE 5).



Figure 2.
Creating a
web service
client from
its live WSDL
document.

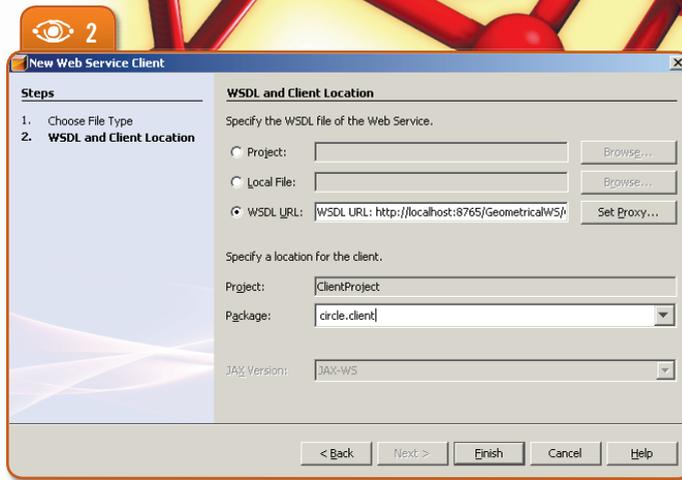
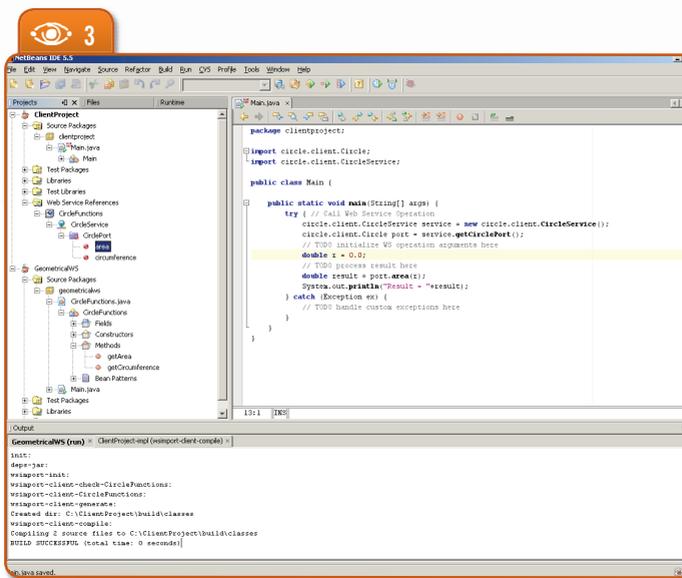


Figure 3.
Creating the
web service
client invocation
interactively.



Listing 2. The Main class, a full web service server.

```
package geometricalws;

import javax.xml.ws.Endpoint;

public class Main {
    public static void main (String[] args) {
        String wsAddress = "http://localhost:8765/GeometricalWS/CircleFunctions";
        Endpoint.publish(wsAddress, new CircleFunctions());
        System.out.println("Web service was published successfully.\n"+
            "WSDL URL: " + wsAddress + "?WSDL");

        // Keep the local web server running until the process is killed
        while (Thread.currentThread().isAlive()) try {
            Thread.sleep(10000);
        } catch (InterruptedException ex) {}
    }
}
```

Create another Java Application project and name it (say) “ClientProject”. If you’re using Java SE 5, add the JAX-WS 2.0 library to the project, as before.

Right-click on the project and choose *New>Web Service Client*. Then fill the *WSDL URL* field with the URL for the web service we just published (see **Figure 2**). Also set the package name for the client artifacts (these are Java classes which will be generated from the WSDL); I used “*circle.client*”. Click *Finish*, and a node named *CircleFunctions* should be created under *Web Service References*.

Open *Main.java* in the source editor, expand *Web Service References* and locate the node *CircleFunctions-CircleService-CirclePort-area*. Then drag it to the editor inside the *main()* method in the *Main* class. NetBeans will generate code that invokes that operation (see **Figure 3**). Next, change the value for the web service operation argument (*r*). **Listing 3** contains the finished source (after tweaking the generated code a little). Finally, run the client project. For *r = 10.0*, the following message should appear in the output:

Robert Eckstein and Rajiv Mordani’s article about JAX-WS 2.0 With the Java SE 6 Platform.

java.sun.com/developer/technicalArticles/J2SE/jax_ws_2

Article comparing various (SOAP) binding styles.

www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/

Result = 314.1592653589793

Deploying the service in a Java EE container

As we've seen up to now, a Java EE application server is not required to bring up a web service. But what if we had to make the service available in a Java EE container? We can just create a "wrapper" Web application that reuses all code written for our Java SE app.

Start by creating a Web app in NetBeans: *File|New Project>Web>Web Application*. Then set its target server to Sun Java System Application Server (if asked). Go to *Properties>Libraries*, click *Add Project* and select the root directory of the *GeometricalWS* project, as in **Figure 4**.

Deploy the web application to the container, with the project's *Deploy Project* action. If you open SJSAS's admin console, the *Circle* web service should be among the listed web services (**Figure 5**).

The *Test* button allows you to test the web service from the admin console (**Figure 6**). After invoking an operation, you can see its request and response.

Conclusions

In this tutorial, we created a web service with just a few lines of web-service specific code (like annotations in the service implementation and publishing in the main class). Then we created a client without manually writing any code – just by using NetBeans' wizards and drag-and-drop features. Finally, we deployed the same web service in a Java EE server, again not having to write additional code. This shows how you can develop web services that are reusable both in Java SE and Java EE environments. ☺

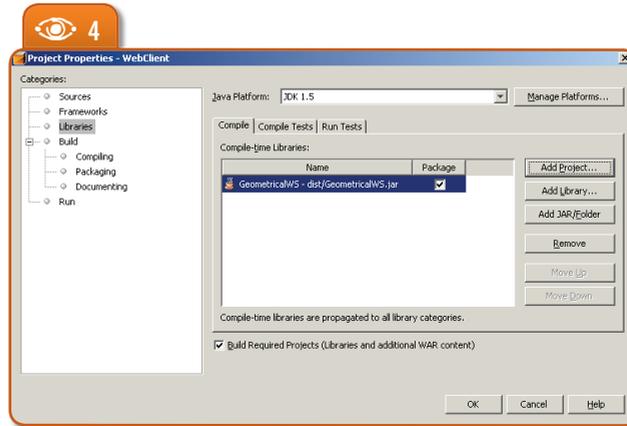


Figure 4. Creating a Web application that depends on the web service project.

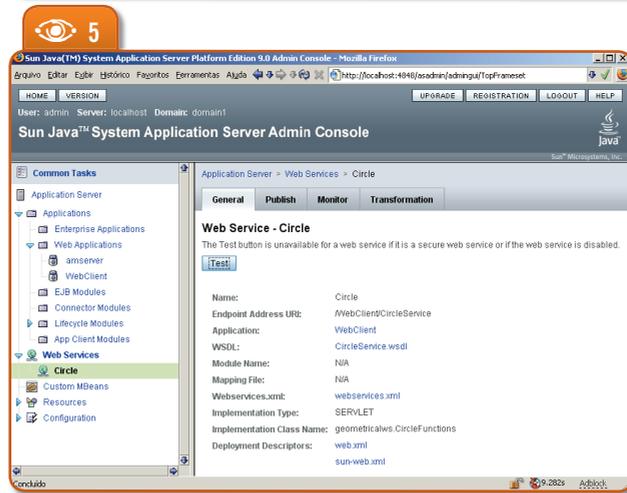


Figure 5. Inspecting the web service, deployed as a Java EE web application in SJSAS.



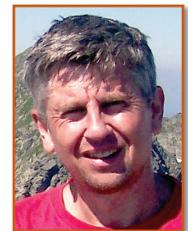
Figure 5. Testing the web service with SJSAS's automatic test page.

Listing 3. The client project's Main class.

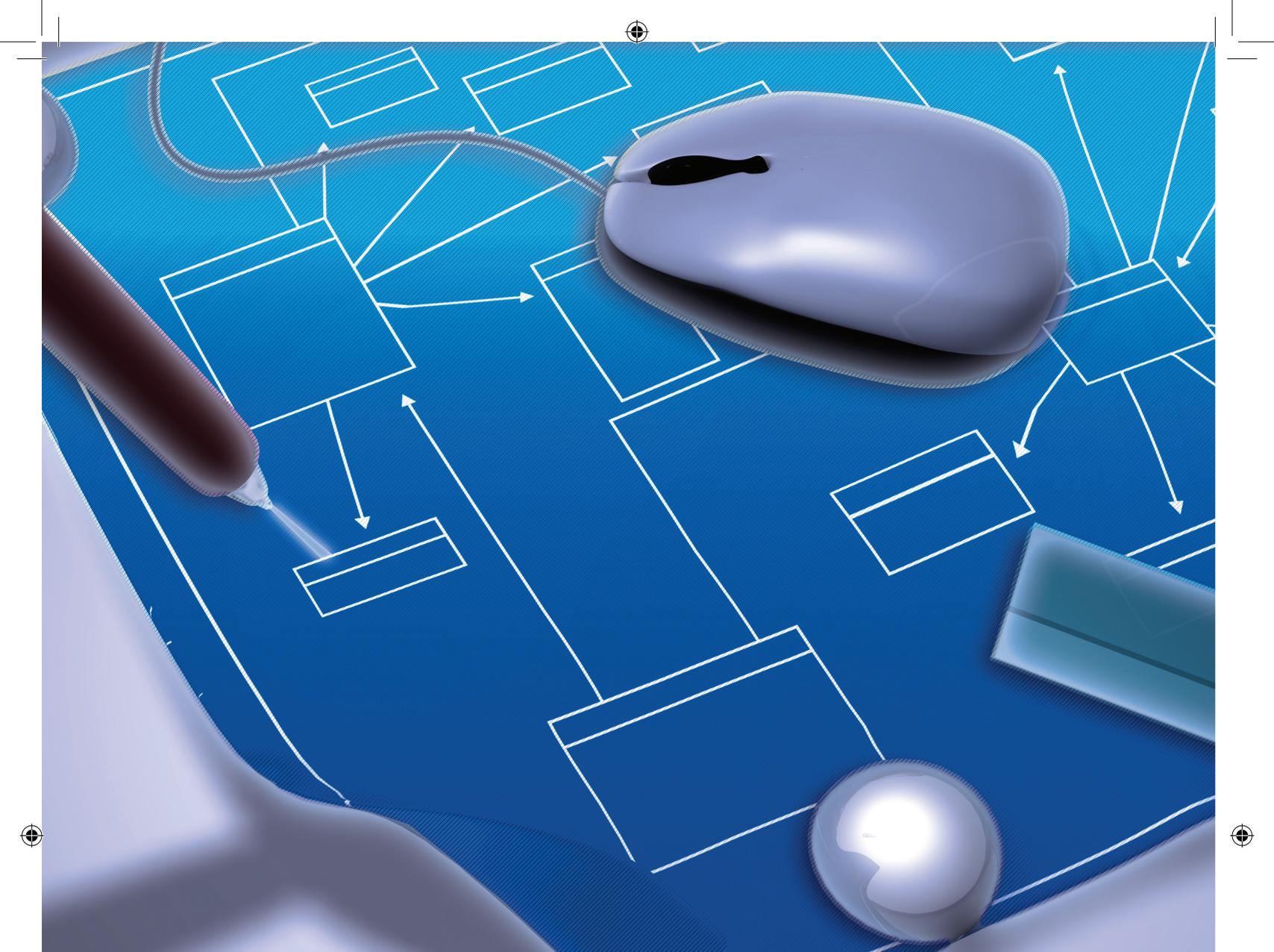
```
package clientproject;

import circle.client.Circle;
import circle.client.CircleService;

public class Main {
    public static void main(String[] args) {
        try { // Call Web Service Operation
            Circle port =
                new CircleService().getCirclePort();
            double r = 10.0;
            double result = port.area(r);
            System.out.println("Result = "+ result);
        } catch (Exception ex) {
            // TODO handle custom exceptions here
        }
    }
}
```



Milan Kuchtiak (*milan.kuchtiak@sun.com*) has worked six years for Sun Microsystems in the NetBeans team, currently on the IDE support for Web application development and Web Services (JAX-RPC and JAX-WS).



Open-Source UML Modeling

**Learn to use the NetBeans UML Modeling Project
by reverse engineering a complete Java application**

Fernando Lozano

The UML Modeling Project (uml.netbeans.org), which started as a part of the Enterprise Pack is now a separate download providing full UML support in NetBeans IDE. This article showcases NetBeans' UML support, going through a case study where we reverse engineer a working Java application, highlighting the IDE's modeling features in the process.

UML and the Java developer

If you are an Agile Modeling advocate like me, your first thought might be “why should I care about UML support in a Java IDE?” Today, leading software engineer authors and even UML early adopters like Martin Fowler and Scott Ambler stand by “modeling on a napkin”: using models only as sketches before building the “real software” (i.e. the source code). Indeed some developers today dismiss modeling as unnecessary and bureaucratic.

However, UML modeling will be useful at some point to most Java developers, whether they're involved in enterprise, desktop or mobile projects. Modeling has its role even in radical XP development, as no significant piece of software can be built with a minimum level of quality without some planning and design. “Traveling light” doesn't mean you should not model at all, but that you should do all the modeling that adds value to your project.

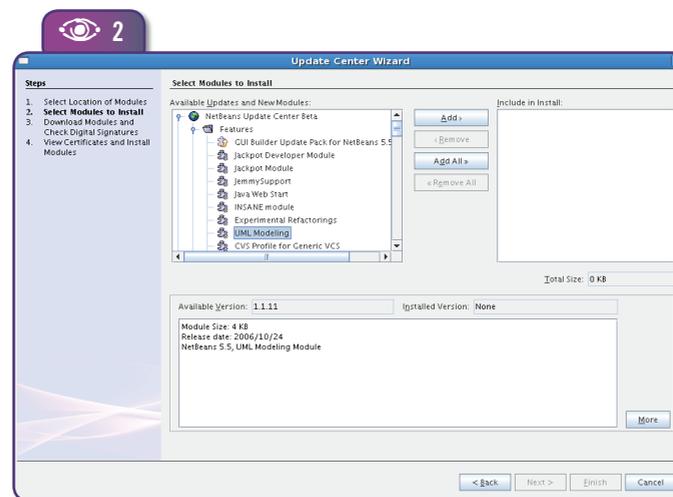
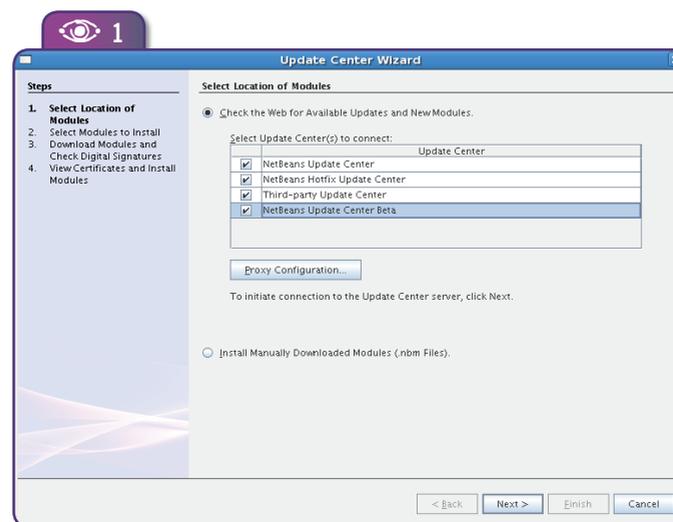
Not all Java developers are Agile Modeling converts though. Many prefer to embrace the MDA paradigm, at the other end of the modeling spectrum. In Model-Driven Architecture the idea is that your models will generate most if not all source code,

and MDA tools like AndromDA (andromda.org) have been quite successful in the Java arena. MDA advocates will be happy to find that the NetBeans UML modules provide many features absent from other open-source CASE tools, and even come with an extensible Design Pattern Catalog, including not only classical GoF patterns but also many Java EE Blueprints out-of-the-box.

Installing NetBeans UML Modeling

UML Modeling is available as a feature from the NetBeans *Update Center* (accessible through the IDE's *Tools* menu). At the time of writing, you had to check the *NetBeans Update Center Beta* (see **Figure 1**) to download it, but readers will probably find the UML Modeling Project on the stable *NetBeans Update Center* when this edition is out.

In NetBeans, select *UML Modeling* in the *Features* category and click *Add* (**Figure 2**). You'll see there are many modules to install



uml.netbeans.org
NetBeans
UML Modeling
Project web
site

Figure 1. Selecting the Update Center for modules still in beta

Figure 2. Selecting the UML Modeling feature

Official
OMG site
uimi.org
dedicated to
UML, where
you can
download
UML speci-
fications and
find about
other UML
CASE tools
and tutorials

Figure 3.
Modules to
install for the
UML Mod-
eling feature

(**Figure 3**). After you agree to a few licenses, NetBeans starts downloading. When the downloads are done, click *Finish* to install the modules (see **Figure 4**). (If you are not the only user of your development workstation, be sure to check the *Global* option for all modules.) Finally, after restarting NetBeans you'll find new UML project templates and configuration options in the IDE.

UML features overview

The NetBeans UML Modeling Project supports ten standard UML diagrams: Use Case, Class, Collaboration, Sequence, Activity, State, Component, Deployment, Object and Robustness.

Even though they're not displayed in the New Diagram wizard, Object diagrams can be built as a Collaboration diagram, and Robustness diagrams as Class diagrams (the standard stereotype icons for Boundary, Control and Entity are provided).

Other features worth noticing are:

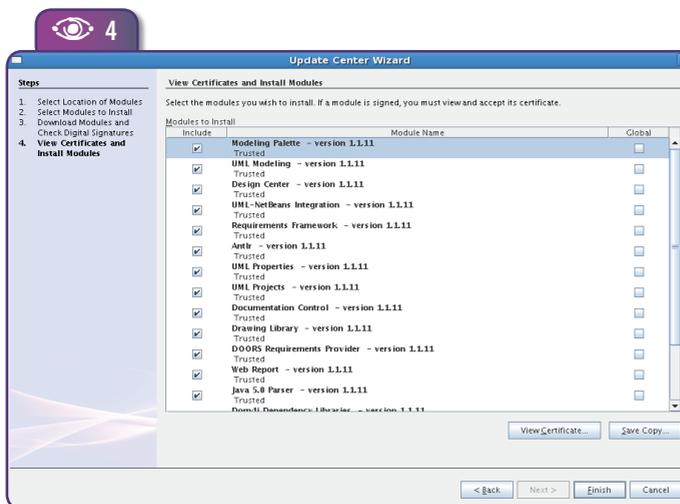
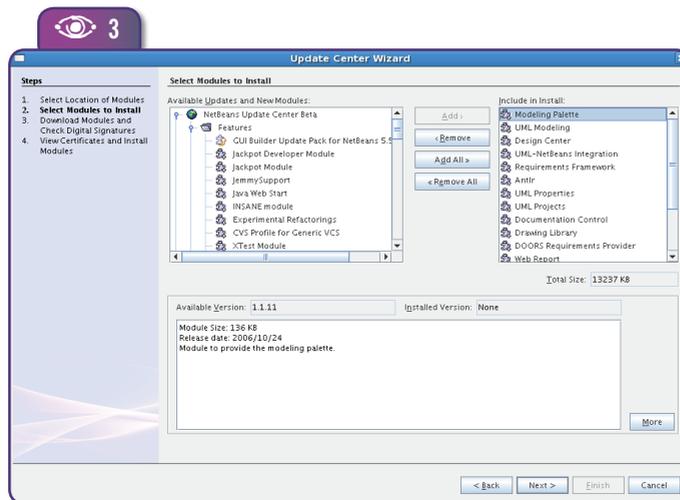
- Java code generation – Java packages, classes and interfaces can be generated directly from the UML model;
- Round-trip engineering – if a UML project is linked to a Java project, changes in code are automatically reflected in the model;
- Ability to nest diagrams and packages inside other model elements;
- A Design Center, which provides an extensible design pattern catalog;
- Rich-text documentation for model elements created directly inside the IDE;
- Operations and attributes can be shown in class elements using either standard UML notation or Java syntax;
- Generation of model reports in JavaDoc format.

The Case Study

We'll use the reverse engineering of an existing desktop application as a case study for exploring NetBeans' UML features. During the process, we also show hints that are useful for forward-engineering projects (from model to source). Readers are not expected to be proficient in UML modeling to follow this tutorial, but basic modeling knowledge will be helpful.

The application used for the case study was created for NetBeans Magazine Issue One. It's a simple Task Manager, similar to those provided in a PIM suite. The "Todo" application has two windows: a task list (**Figure 5**) and a task editing form (**Figure 6**). The task list window displays tasks ordered either by priority or by due date, and it can filter out completed tasks. Tasks are colored to indicate completed, late or alert status. The complete application is available as a download in the NetBeans Magazine's website.

Figure 4.
Click *Finish*
to install
all UML
modeling
modules



Our application is quite simple, so reverse engineering it won't be a complex task. Even so, it is complex enough for us to demonstrate many of NetBeans' UML features. We'll work through the application as if we didn't know how it was designed, using UML diagrams to understand its structure and design ideas.

Creating the reverse-engineering project

Installing the UML modeling project adds a new project category to NetBeans, with three project templates (**Figure 7**).

Platform-Independent Model projects use standard UML notation for class attributes and operations, and do no code generation. This template is useful in the initial stages of the development process, when all models are “conceptual” and not “physical”.

Java-Platform Model projects use Java syntax for class attributes and operations and must be linked to a Java Application or Library project. NetBeans will keep both projects in sync: If you edit the code, the UML model will change to reflect new or changed classes, attributes and operations, and if you change the UML model, the source code will be modified accordingly.

We'll use the third project template, which populates a Java-Platform Model from an existing Java project. Click the *New Project* icon and choose the UML category. Then select *Java-Platform Model by Reverse Engineering a Java Project*. Accept the defaults for the next step, except for selecting the *Todo Java* project; then click *Finish* (see **Figure 8**). NetBeans will scan

the sources and populate the project with model elements for Java packages, classes and interfaces. (Note that for large projects this may take a long time and eat up a large amount of memory.)

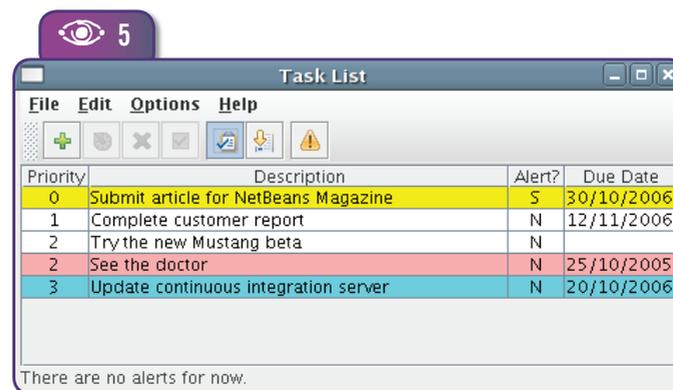
 If you experience *OutOfMemory* errors while reverse engineering, you can edit the file *etc/netbeans.conf* inside your NetBeans installation and change the value for `-Xmx`, increasing the maximum heap size from 128M to 256M or bigger.

While scanning, the IDE shows a log window detailing the progress for each source file. When the scanning finishes, click *Done* to dismiss this window, and you'll end up with a new UML project as shown in **Figure 9**.

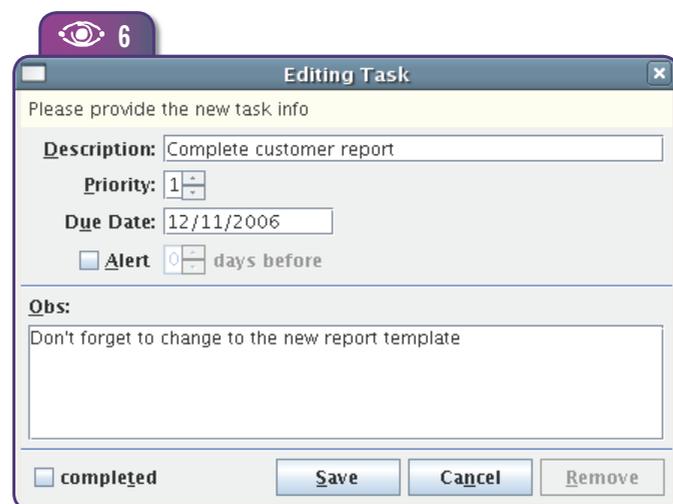
Anatomy of a UML Project

A UML project is organized into three containers: *Model*, *Diagrams* and *Imported Elements*.

The *Model* container holds all elements created as part of the model. Elements can be Classes, Interfaces, Packages, Actors,



 **Figure 5.**
The task list window



 **Figure 6.**
The task form window

 How to use UML as part of a traditional software development process

The Unified Process Explained, Kendall Scott

Figure 7.
UML project
templates

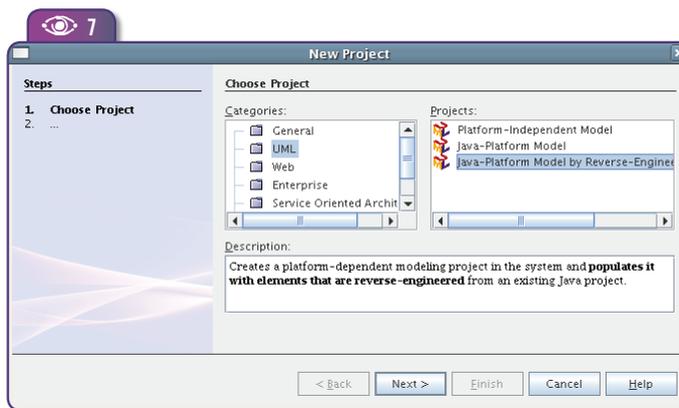


Figure 8.
Creating a
UML Project
for reverse-
engineer-
ing a Java
Project

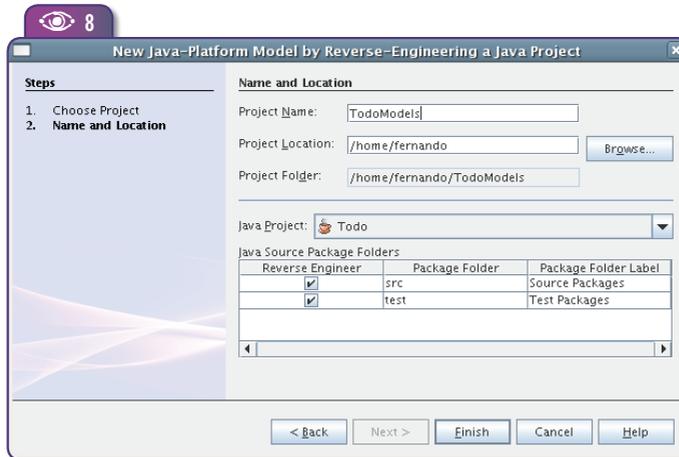
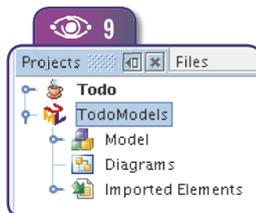


Figure 9.
The new UML
project



Diagrams, Notes, Data Types, Associations and anything else that can be defined as part of the UML project itself.

The *Diagrams* container provides quick access to all diagrams created as part of the model. You may be surprised to see there are no diagrams in our reverse-engineering project. That's because a UML model consists of *model elements*, which may or may not be shown as diagrams. Every UML diagram is a partial view of a model, and you'll need to choose which details to show and which to leave out. So NetBeans populates the reverse-engineered model with elements but leaves the diagram creation to the developer.

Finally, the *Imported Elements* container allows a model to reference elements created as part of another model (i.e. another UML Project). This is useful for organizing large models as a set of UML projects; for example one for business logic and another for GUI elements. It's also useful for reusing model elements in various projects. And it allows for a Platform-independent UML Model that's referenced by a Java Platform UML Model. This way, you can start

with conceptual modeling and then move on to physical modeling, and keep track of which physical elements realize which conceptual elements¹.

Exploring the reverse-engineered model

Expand the *Model* container and you'll see that it includes both application-specific and standard Java packages and classes (**Figure 10**). As a UML model cannot reference anything it does not contain or import, NetBeans creates model elements for Java SE classes. (An alternative would be having a UML model with the Java SE API and import elements from it.)

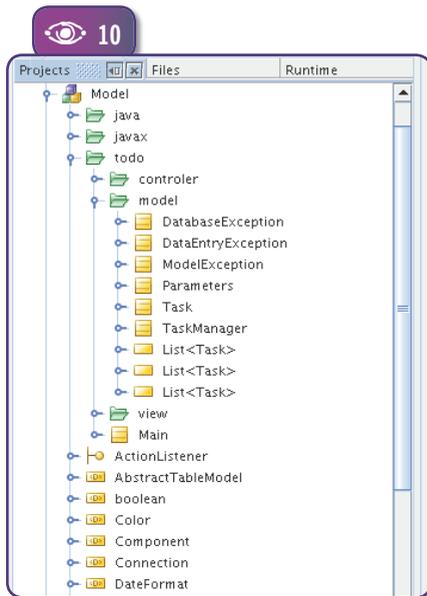
Sometimes NetBeans will create duplicates, like the many **List<Task>** elements you see in **Figure 10**. It's better to leave them as they are, as removing them may delete attributes from the Java project.

Generating a class diagram

The *Todo* application has only a few classes, all of which are inside the **todo** package and its subpackages. It would be possible to create a "complete" Class Diagram, but for most real-world applications the resulting diagram would be very hard to read and therefore mostly useless. It's usually better to start by creating a class diagram for each package, and then check the dependencies between packages or between a few important classes.

Let's start with the **todo.model** package. Select all its elements, right click on the selection and choose *Create Diagram from Selected Elements* from the context menu. Then choose *Class Diagram* in the wizard (see **Figure 11**).

¹ A «realize» UML relationship between two model elements states that one element represents an abstract concept in the problem domain, while the related element represents a software construct implementing that concept, making it "physical" in the software's point of view.



Note that the “Namespace” combobox lists what seem to be all the packages in the current model. In fact, every UML package (and most other UML elements) corresponds to a *UML Namespace*. You can use namespaces for any kind of model element. Leaving the class diagram for the `todo.model` package inside the `todo::model` namespace makes the diagram easier to find. It also makes it clear that the diagram refers to elements of that package.

The resulting diagram is shown in **Figure 12**. Note that NetBeans is pretty smart in laying out its elements. But even with an optimized layout and a small set of classes, the generated diagram is quite large. The fact is there’s too much detail, especially if we just want to understand the relationships between elements in the package. Fortunately you can hide most of the excess information.

A good start is hiding all private fields and methods (**Figure 13**). Right-click on each class and select `Compartment > Customize`, then uncheck the entire “Attributes” com-

partment or expand it to select specific fields to hide. After hiding an element’s details, you can right-click `Resize Element to Contents` to optimally resize it (**Figure 14**). (You can also hide/show a compartment by double-clicking the compartment separators.)

After these changes, the diagram will probably need some rearrangement. NetBeans can do this using four different auto-layouts, accessible through the last four buttons in the Diagram Editor toolbar (see **Figure 15**). Click the first button, for Hierarchical Layout. This is the default layout NetBeans uses when generating a diagram from model elements. The results are shown in **Figure 16**.

Inferring meaning from a generated diagram

Notice I deleted the `List<Task>` element from the diagram, as well as all exception classes. When deleting an element, NetBeans gives you the choice of removing it from the diagram only, or from the diagram and the model (see **Figure 17**).

I’ve shown only public methods for `Parameters` and `TaskManager`,

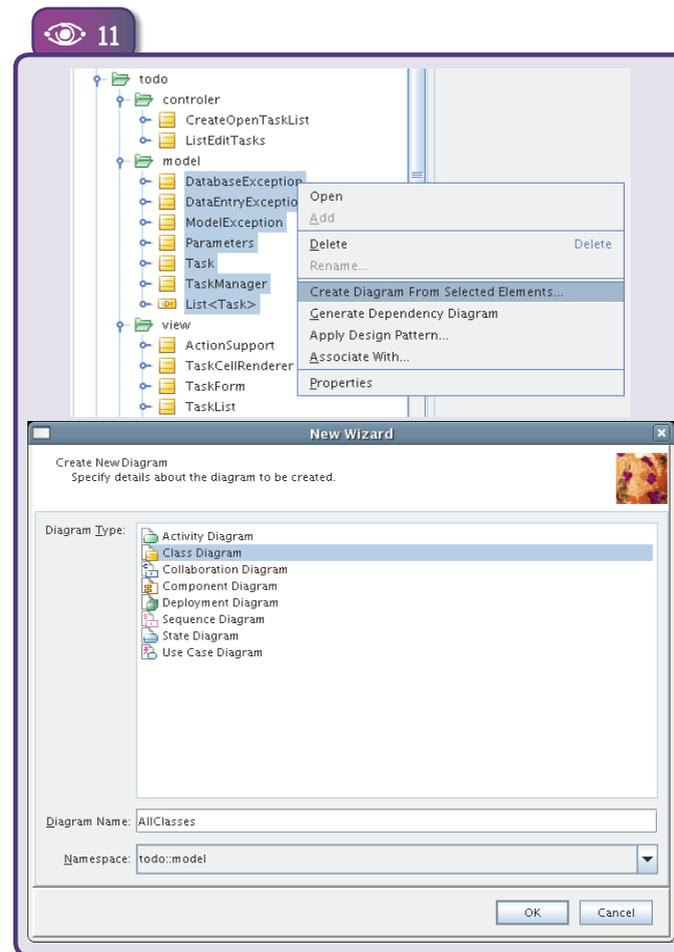


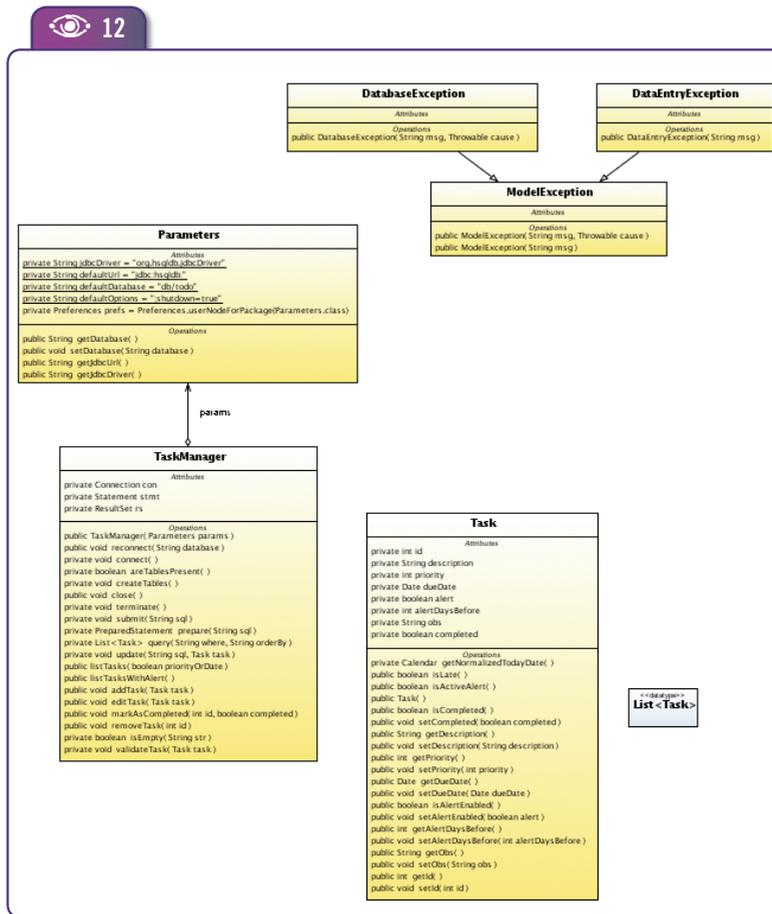
Figure 10. The new UML project, showing a few model elements

Figure 11. Creating a class diagram from model elements

Articles about UML, modeling, application design and agile modeling

martinfowler.com/articles.html
objectmentor.com/resources/publishedArticles.html
thoughtworks.com/byline-articles.html

Figure 12. The NetBeans-generated model for all elements inside the *todo* model namespace.



The UML standard, by its creators
UML User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson

Figure 13. Hiding details from classes

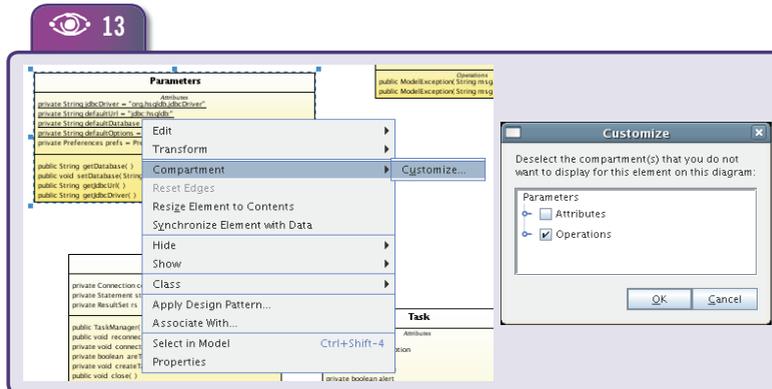
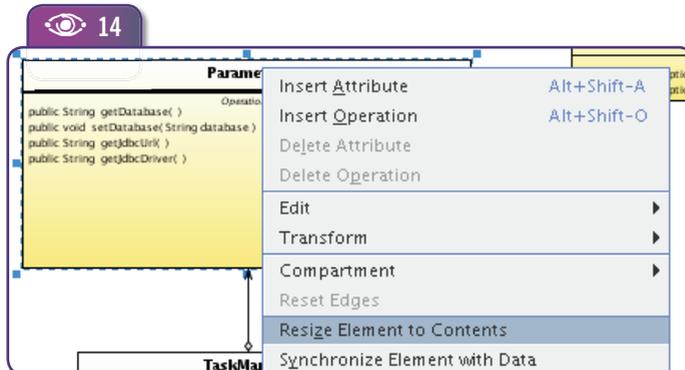


Figure 14. Optimally resizing a class



and only attributes (all private) for the **Task** class. That's because I assume **Task** is a DTO (Data Transfer Object) or Value Object, and showing accessor methods wouldn't help in understanding the class. But for **Parameters** the attributes do not map exactly to the getters, so I assume the class has some intelligence of its own. I also assume **TaskManager** is a DAO class, and that it uses **Parameters** to get JDBC connection parameters.

As you can see, reverse-engineering a model from Java code involves a lot of assumptions and common sense. You'll always be making educated guesses about the role played by each element in the application, and using these decisions to create your diagrams to best effect.

Enhancing the generated model

Based on the reverse engineering done so far, we have identified the use of common design patterns in the application. UML provides a nice feature to make these explicit in a model: stereotypes². A few standard stereotypes are defined in the UML standard, and you can create new ones freely. We'll

create the «VO» and «DTO» stereotypes so our interpretation of the purpose of each class is explicit in the model.

To add a stereotype to a class, use the properties window, as shown in **Figure 18**. An editor allows you to select existing stereotypes or create new ones. Note that an element can have many stereotypes, although this is not common.

Now, if our conclusions are OK, the model generated by NetBeans is not entirely correct. There is no part/whole relationship between **TaskManager** and **Parameters**, so they should not be connected by an aggregation. This should be changed to a simple association. Right-click on the aggregation (it should turn blue) and select *Transform>Remove Aggregate* from the context menu (see **Figure 19**).

In fact, you should expect NetBeans (or any other UML CASE tool) to generate a few “incorrect” associations when reverse engineering Java sources. That’s because UML associations have much richer semantics than Java references. All structural associations between Java classes originate from attributes referencing other classes/interfaces (or collections and arrays of these). Similarly to most other programming languages, Java does not distinguish between composites, aggregates and simple associations. Also, a reference to an instance of another class could be there just for convenience, and not be a structural association at all.

Now, if **TaskManager** is a DAO class, responsible for persisting instances of **Task**, it’s clear that these two classes are strongly coupled. Changes in **Task** fields will probably require changes in **TaskManager**’s behavior. This can be made explicit in a UML diagram by a *Dependency* relationship, which is available in the UML Class Diagram Palette in the *Dependencies* category. Click on the *Dependency* tool, then on **TaskManager** in the diagram and on **Task** (also in the diagram). This creates a dependency connecting the two classes. The end result is shown in **Figure 20**.

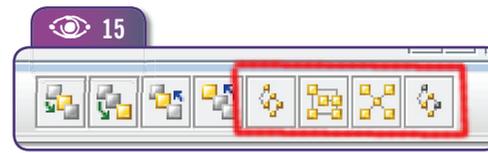


Figure 15. Auto-layout diagram buttons from the NetBeans diagram editor toolbar

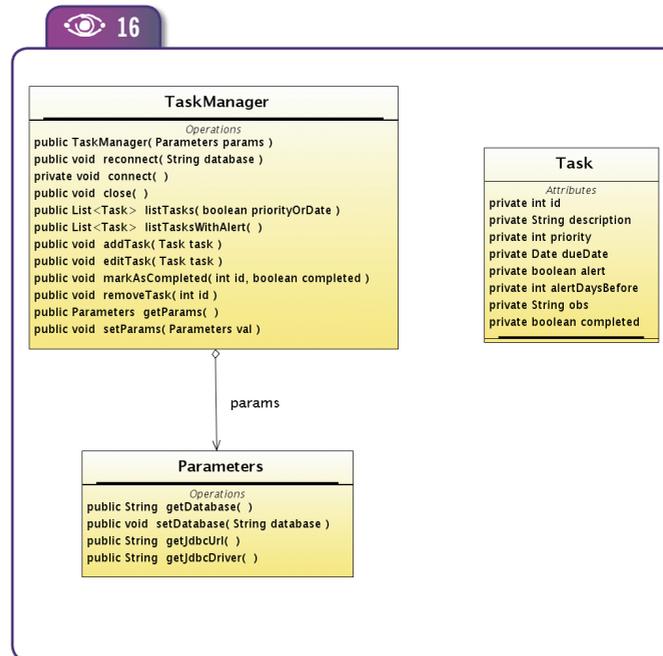


Figure 16. Class diagram for the *todo.model* package, after hiding details, deleting elements and applying an auto-layout

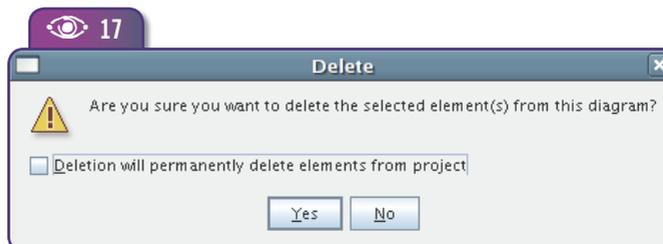


Figure 17. By default, deleting an element from a diagram doesn’t remove it from the model

Generating Dependency Diagrams

From our understanding so far, it looks like **TaskManager** is the main class of the **todo.model** package. But does our model show everything important about this package and about **TaskManager**?

NetBeans helps us check this by generating a dependency diagram, a type of class diagram that shows everything that’s referenced by a given class. Right-click a class in either the Projects window or the class diagram and select *Generate Dependency Diagram*. The diagram generated for **TaskManager** is shown in **Figure 21**. It’s created inside the namespace defined by the class itself, so it shows in the Projects window as a child node of the class.

You can see that the only elements in the diagram that are not in our **todo.model** class diagram come from the Java SE APIs. You’d

² A UML stereotype is a label you can add to elements to highlight specific semantics. You can think of a stereotype as a subclass or role of a kind of model element.



Figure 18. Adding a stereotype to a class



Figure 19. Transforming an aggregation into a simple association



Figure 20. Class diagram model for the `todo.model` package after changing associations between elements. Note the dependency from `TaskManager` to `Task`, and that the association from `TaskManager` to `Parameter` is no longer an aggregation. Note also the stereotypes for `TaskManager` and `Task`.

get similar results with other classes from the same package, so it's safe to assume the class diagram built for the package has enough information.

The dependency diagram for `TaskManager` is an example of a “disposable diagram”. There's no value in keeping it around after using it to verify that other related diagrams have enough information. You should delete the diagram from the model, and also the dependency diagrams for other classes in the same package (if you generated them). The key is not to pollute your model with useless diagrams.

NetBeans-generated dependency diagrams don't take into account classes referenced inside a method body. Only attribute and method declarations are followed, so there will be cases when you'll need to inspect the code to find additional dependencies and add them manually.

The `todo.controller` package

The `todo.controller` package has only two classes. A class diagram for this simple package would be of little use, but let's see what a dependency diagram tells us. **Figure 22** shows the dependency diagram for the `ListEditTask` class, already “simplified” as we did with the `todo.model` package class diagram.

Here we see many classes coming from different packages. Either we have found a crucial class for the application, or the application design is a mess. Method signatures may help us understand how these classes are related to each other. They suggest `ListEditTasks` is an event listener for both `TaskList` and `TaskForm`. It looks like `ListEditTasks` reacts to those events by calling `TaskManager` methods, acting as a workflow controller for the application.

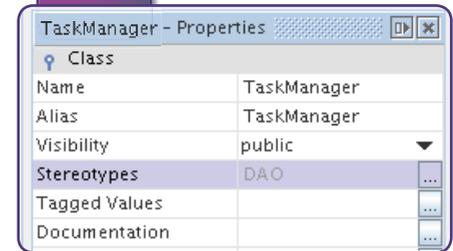
Note also that `TaskList` and `TaskForm` are the two windows in the application. The fact that both are connected to `ListEditTasks` strongly suggests it's the main controller class for the application.

The dependency diagram for `CreateOpenTaskList` shows connections to `TaskList` and `TaskManager`, similar to the ones from `ListEditTasks`. It seems `CreateOpenTaskList`

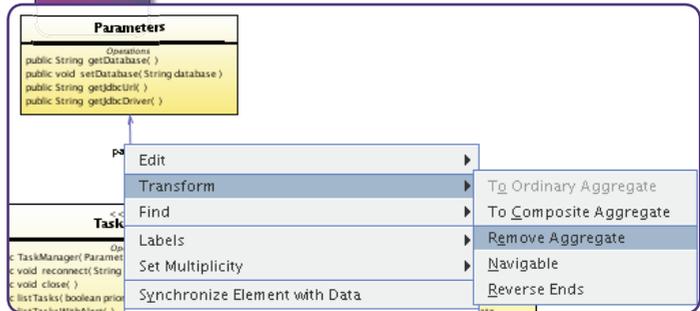
is the controller for some secondary usage scenarios, so we should examine `ListEditTasks` the same way we did with `TaskManager`.

If we have found the main controller class, the dependency diagrams for the window classes should not add much new information. Surprisingly, the dependency diagrams for `TaskList` and `TaskForm` do not show any connection with `ListEditTasks`. But they do show a new connection with `Action-`

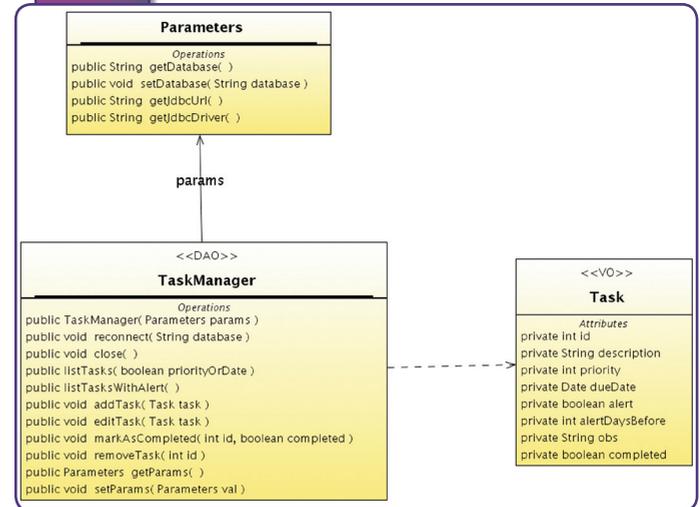
18



19



20



Support, which is a Swing **ActionListener**. **ListEditTasks** is also an **ActionListener**, so there may be some indirect connection from the window classes to the main controller class.

Event-driven programming is meant to reduce coupling between classes, and it succeeds so well that there's no way, just by analyzing Java method signatures and

attribute declarations, to find a connection from **TaskList** (or **TaskForm**) to **ListEditTasks**.

A diagram for the application UI

Figure 23 shows what we've learnt (and guessed) so far about the Todo application user interface. The diagram was built by merging information from **ListEditTasks**, **TaskList** and **TaskForm** dependency diagrams.

Starting with the **ListEditTask** dependency diagram, I added the **ActionSupport** and **Task** by dragging these from the Projects window to the diagram area. NetBeans automatically adds connections to others elements in the diagram. Then I simplified the classes' presentation, corrected associations and added new dependency links, as done before for the **todo.model** package class diagram.

The finished diagram is more than just a **ListEditTask** dependencies diagram, so I renamed it to **CoreTodoDiagram** and moved it to the **todo** package. It gives us a pretty good picture of the

application structure, but still doesn't help understanding how the **ListEditTasks** class actually reacts to user interface events from **TaskList** and **TaskForm**. We could guess **ActionSupport** has a role in this, but so far we don't know exactly what role. Perhaps it's a utility class that could be deleted from the diagram...

The fact is, UML class diagrams are good to show static structure, but not dynamic behavior. Fortunately, the UML provides other diagrams that help us understand how an application behaves at runtime.

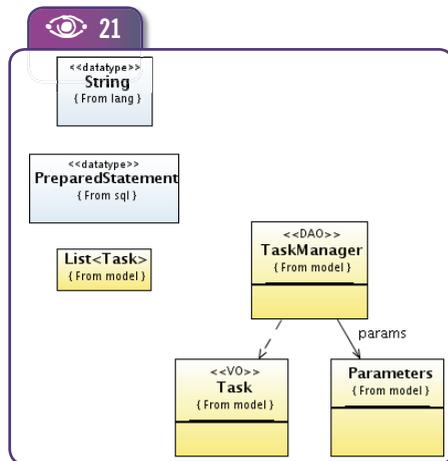


Figure 21. Dependency Diagram for TaskManager

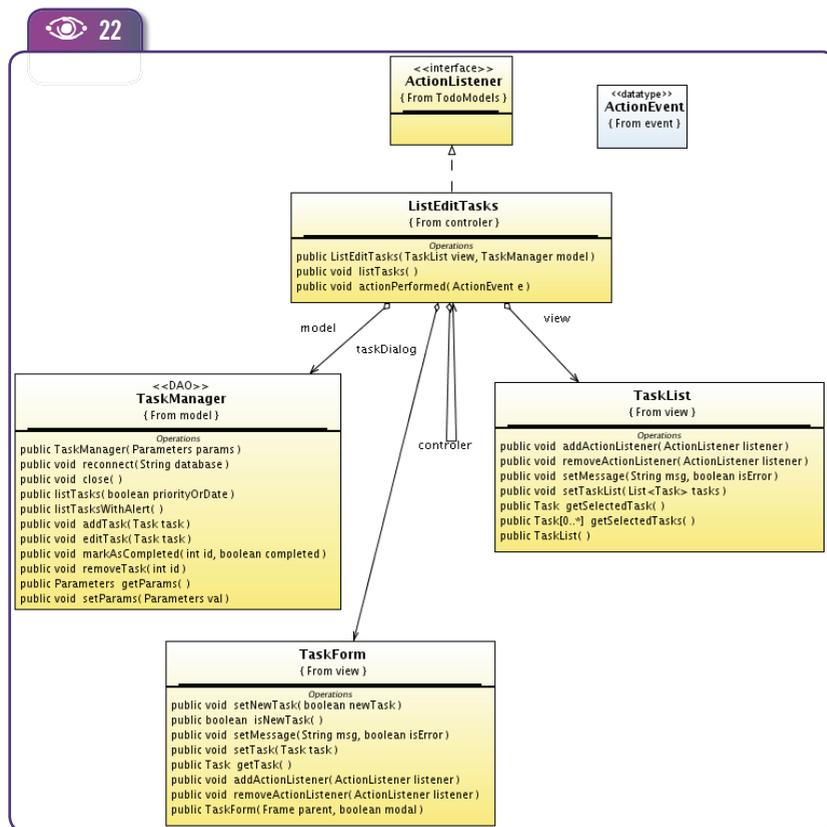
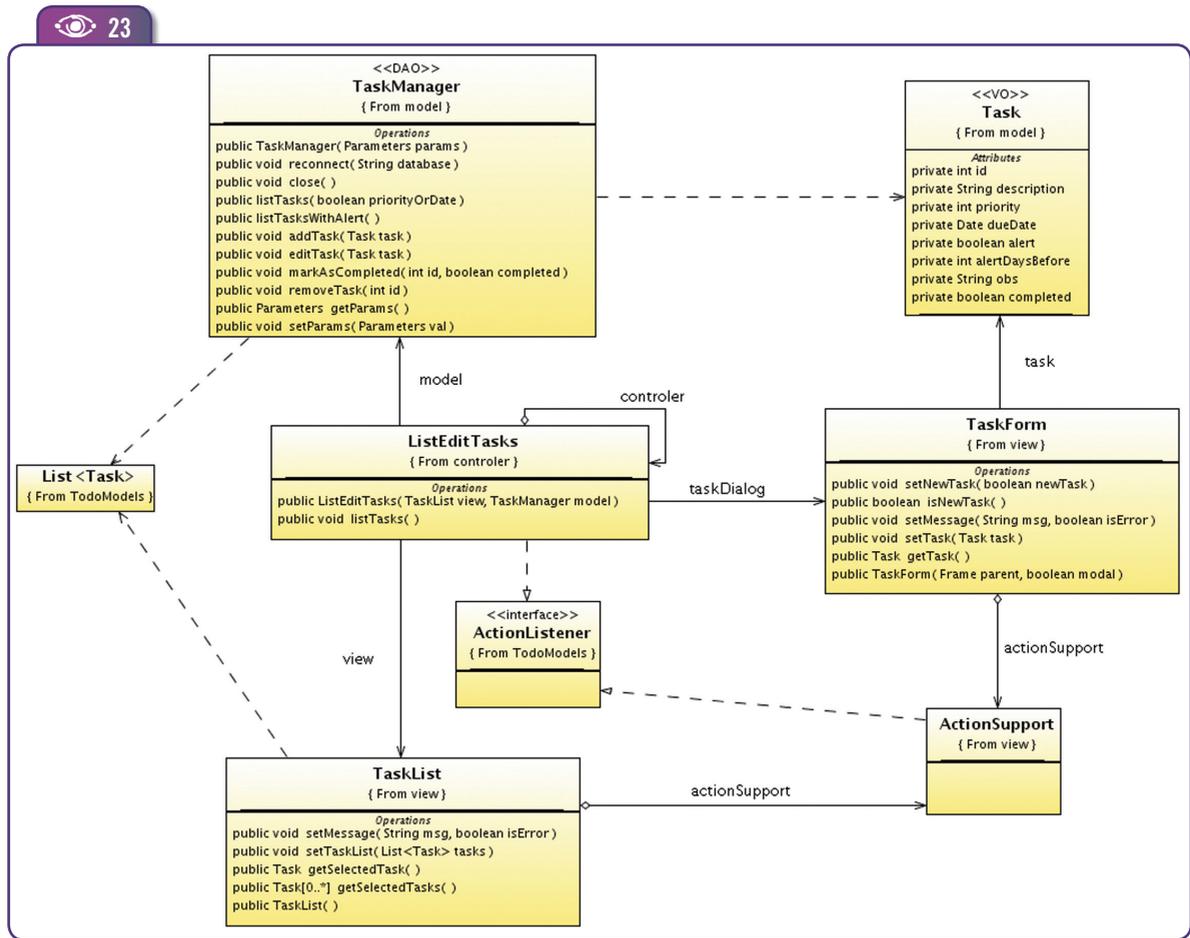


Figure 22. Dependency Diagram for ListEditTasks

Extreme Programming Explained, Kent Beck If you want to use an Agile Modeling process instead of a “traditional” one

Figure 23.
Class diagram
for the Todo
application user
interface



Generating Sequence Diagrams

We've been working under the assumption that the window classes (**TaskList** and **TaskForm**) generate Swing **Action** events, which are somehow dispatched by **ListEditTasks**. **ListEditTasks** should register itself as an **ActionListener** for **TaskList**, and then **TaskList** should fire **Action** events to **ListEditTasks**.

Note that **ListEditTasks** has only a constructor that receives a view (**ListTasks**) and a model (the DAO class, **TaskManager**) as parameters. Looking at the constructor code in the source Java project we can see this listener registration happening.

Who calls the **ListEditTasks** constructor? The application **main()** method is a good candidate. Expand the **todo.Main** class node to show the **main()** operation (UML calls "operation" what Java calls "method"), then right-click and choose *Reverse-Engineer operation*. In the wizard chose a Sequence Diagram and accept all defaults. The result will be a new diagram node under the operation's node. **Figure 24** shows the diagram that NetBeans creates.

An UML sequence diagram shows the message flow among a set

of objects during the execution of an operation. Note that it refers to class instances and not to the classes themselves. A solid arrow represents a message, that is, a Java method call (in UML terms, an operation call). A dashed arrow is either a message return or a "create" message, i.e. the creation of a new object.

Again, the automatically generated diagram comes with too much detail. Removing the **args**, **length**, **err** and **System** objects, and moving the elements around should solve this (see the result in **Figure 25**).

The diagram shows that **main()** just constructs instances of the classes we have already identified as being the main ones for the application, and that all are connected by the **ListEditTasks** constructor. So

Some freely available UML tutorials
bdn.borland.com/article/0,1410,31863,00.html
sparsystems.com.au/UML_Tutorial.htm



far we've gleaned the same information we had expressed in the class diagrams. But it's important to confirm our guesses.

The fact the `main()` operation creates no instances of other classes, and that it doesn't call operations other than constructors and `setVisible()` for the `TaskList` window, makes us confident of not missing any important class.

Interpreting sequence diagrams

The `main()` method sequence diagram shows connections made from `ListEditTasks` to `TaskManager` and `TaskList` (we're ignoring the connections from `CreateOpenTaskList` to `TaskList` and `Parameters`). But the diagram doesn't show connections going the other way, that is, from `TaskList` to `ListEditTasks` to dispatch `Action` events. We know a view class has to send events to its controller, and that a model class has no knowledge about its controller – so there are no other missing connections.

If you follow the code, you'll see that the `ListEditTasks` constructor calls `TaskList`.

`addActionListener()`, confirming that `ListEditTasks` handles `Action` events generated by `TaskList`. This is the missing connection from `TaskList` to `ListEditTasks`.

NetBeans won't follow a nested class from one operation to another. So the message `addActionListener` from `ListEditTasks` to `TaskList` is missing in the sequence diagram. It's helpful to add this message manually, to get a complete picture of all connections between the `Todo` application main classes.

NetBeans also won't create a sequence diagram from multiple operations (as it can from multiple classes). If it could, a sequence diagram generated from both `main()` and the `ListEditTasks` constructor would show all connections that are being made. That's understandable, as a sequence diagram following nested calls would probably be very big (how many levels down are enough?). It would also probably create a diagram where one operation "sees" the inner workings of others, violating OO encapsulation.

We still need a diagram showing event handling. We also need to clarify the role `ActionSupport` plays in the `Todo` application. Again looking at the code, we can see the window classes delegate all `Action` event dispatching to the `ActionSupport` class. This is done so the window classes themselves don't have to manage multiple event listeners. This way `TaskList` can send `Action` events to both controller classes, `ListEditTasks` and `CreateOpenTaskList`.

By digging a little deeper into the code, we can see that each button or menu item from the window classes provides a different `actionCommand` string. The controller classes use this property to know which operation was requested by the user and then execute

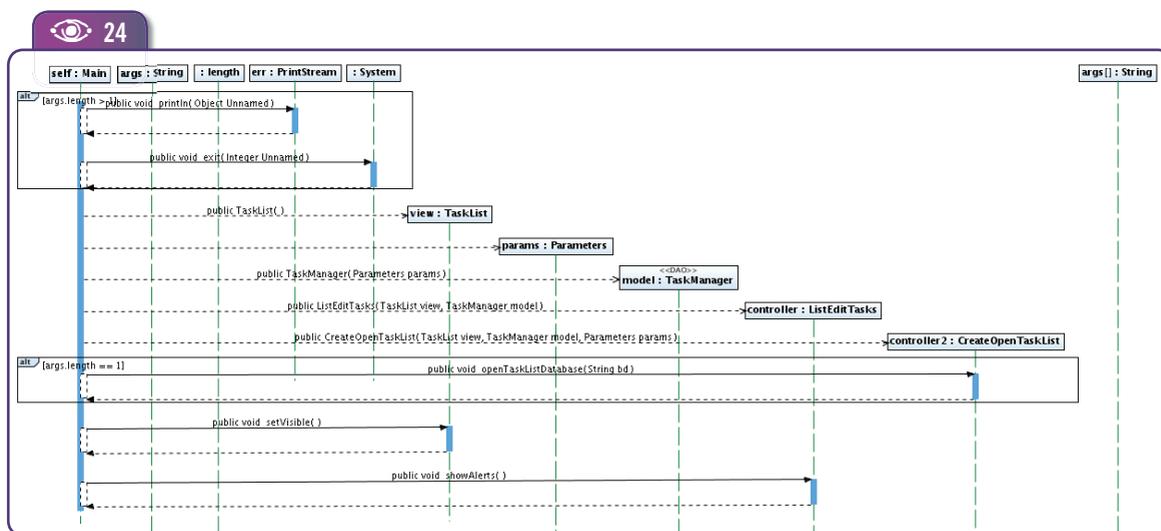


Figure 24. Sequence diagram generated by NetBeans for the `main()` operation

it. So they don't need to know which Swing component (from the window classes) generated which event. This isolates the controller classes from the internal structure of the view (window) classes.

A hand-made diagram

We can't fully automate reverse engineering, but we can document the knowledge gleaned from the reverse-engineering process in the form of UML diagrams, and communicate it

more clearly to other developers. If you are serious about reverse engineering applications, you'll inevitably have to create some diagrams on your own.

A "cross-level" sequence diagram is useful in understanding complex interactions among objects – like event handling for the **ToDo** application, which involves **TaskList**, **ListEditTask**, **ActionSupport** and **TaskManager**. NetBeans makes it easy to create such diagrams.

First, select the **todo** package, right-click and select **New Diagram**. Choose **Sequence**

Figure 25. Sequence diagram for the *main()* operation with excess detail removed

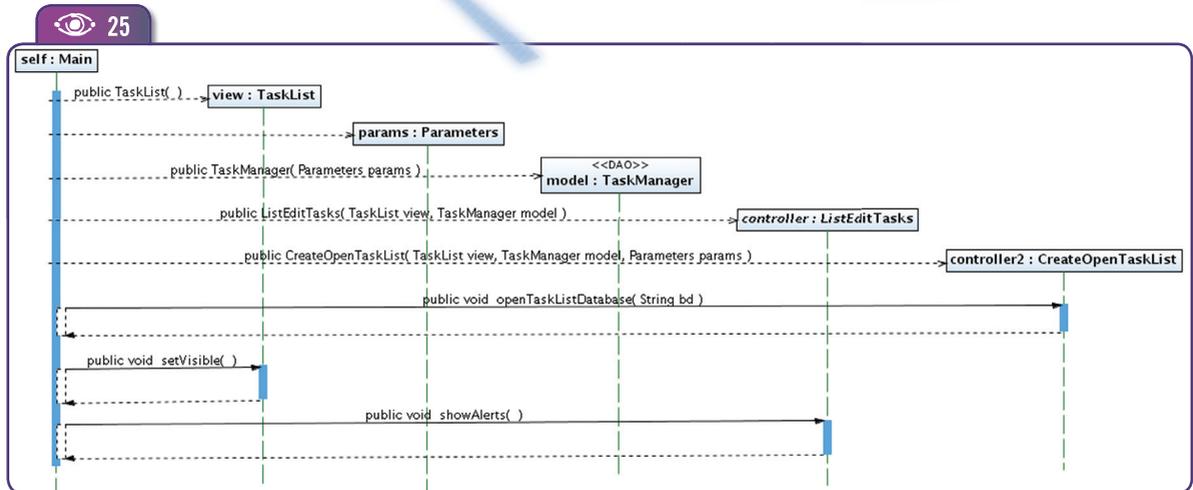
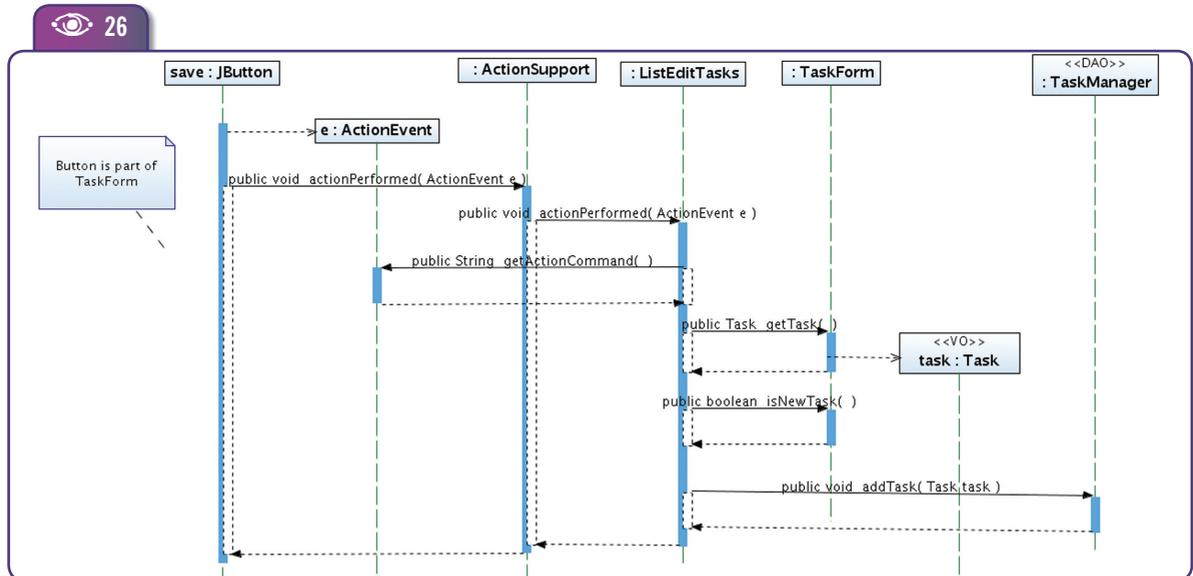


Figure 26. Sequence diagram for the event handling between view and controller classes



Diagram, and name it *ActionEventSequence*.

From the Projects window, drag the elements **JButton** (`javax.swing`), **ActionEvent** (`java.awt.event`), **ActionSupport** (`todo.view`), **ListEditTasks** (`todo.control`), **TaskForm** (`todo.view`), **Task** (`todo.model`) and **TaskManager** (`todo.model`).

Connect these from left to right, using either the *Synchronous Message* or the *Create Message* from the Sequence Diagram Palette. Then, for each message, right-click it and select *Operations* from the context menu to see all operations supported by the message's target. The sequence of messages is *create*, **actionPerformed()**, **actionPerformed()**, **getActionCommand()**, **getTask()**, *create*, **isNewTask()**, and finally **addTask()**. The end result should be like **Figure 26**.

The diagram shows how the core classes collaborate to perform a user action. As a sequence diagram should describe a specific sequence of messages, we chose the Save button (a **JButton**) from **TaskForm** when it's used to add a new task.

Clicking Save starts the sequence, and the resulting event is routed by **ActionSupport** to **ListEditTask**, which queries the **actionCommand** property from **ActionEvent** object (**e**) for which operation to perform. Then it asks the **TaskForm** for the updated **Task** object, which is sent to the **TaskManager** to be saved to the database.

Although this sequence diagram was created to describe a specific message sequence (a click on the Save button)

it is representative of all other

Action events for the Todo

application. There's no need to

build sequence diagrams for each remaining button or menu item. If a developer understands this diagram and how the Save button **Action** event is handled by the application, he can figure out how all other **Action** events are handled.

This ends our case study. We now have a model rich enough so any developer can understand and maintain the Todo application. We could consider drawing other UML diagrams, like Use Case or Deployment, but these would probably add little value in our case. An important part of doing good modeling is knowing when to stop modeling.

Conclusions

In the past, modeling was done using expensive proprietary tools with steep learning curves. This prevented most development teams to effectively use UML models and hindered adoption by most small and medium shops. Modeling was performed mainly by "Business Experts" and "System Analysts", sometimes by a "Software Architect". And many of those never came close to real Java source code. Also CASE tool developers were often unaware of the state-of-the-art in enterprise software development in general and the Java platform in particular.

All this is changing with the full open-source UML support in NetBeans. Having UML modeling seamlessly integrated in the IDE encourages developers to use models and diagrams, improving both communication between team members and the quality of OO designs. The UML Modeling Project provides developers, from Agile Modeling fans to MDA advocates, with a powerful toolset that enables ubiquitous and effective modeling. ☺



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

Introducing the NetBeans Strategic Partner Program

While NetBeans is a powerful IDE and platform all by itself, companies adding new features and capabilities via plug-in modules amplify the value to developers

Matt Volpi



NetBeans Strategic Partner

As an open-source project, NetBeans relies on contributions and feedback from the community to consistently meet the needs and requirements of software developers in a number of disciplines. Companies that have committed to supporting NetBeans by building plug-in modules and recommending NetBeans to their developer communities now have a new set of benefits

for their contribution to the enhancement and growth of NetBeans.

Program benefits

The NetBeans Strategic Partner Program offers a unique set of technical and

co-marketing benefits to qualifying companies. Strategic Partners get access to technical support, the opportunity to sponsor and present at NetBeans events, access to 4.5 million developers via the Sun Developer Network, and *NetBeans.org* web, e-mail properties and other co-marketing opportunities. Strategic Partners are also invited to exclusive briefings on NetBeans plans and activities.

To qualify, Strategic Partners need only meet the criteria of building a NetBeans plug-in and/or recommending NetBeans to a large external or internal developer community. Individuals and companies that use and endorse NetBeans are eligible to become NetBeans Community Partners. This provides a mention on the *NetBeans.org* web site as well as usage of the NetBeans Community Partner logo.

Meet some of the Strategic Partners

New Strategic Partners are joining the program all the time and are featured on the NetBeans web site. Here are some of the companies that have already jumped on the NetBeans bandwagon and are helping NetBeans grow.

Sprint

Sprint Nextel (*sprint.com*) is one of the largest U.S. Mobile operators and has been a pioneer in bringing mobile Java services to its business and consumer customers. To help its application developer community target its wide range of devices in its portfolio, Sprint has worked with Sun Microsystems to create a customized version of NetBeans, NetBeans Mobility

Pack and the Sun Java Wireless Toolkit. This package, called the Sprint Mobility IDE, is available from the Sprint Application Developer Program site.

InsiTech

InsiTech's (*insitechinc.com*) emerging Java technology called XTT is an application development framework for rapid creation of rich, ultra-thin, Java interfaces. Solutions developed in XTT are distributed applications that can be deployed and run without any modifications from LAN to the Internet, on any operating system, and use a variety of data sources (relational, object-oriented, multi-value, etc.). XTT is a 100% pure Java application and plugs into the NetBeans IDE.

Sony Ericsson

Sony Ericsson (*sonyericsson.com*) is a top mobile device manufacturer and has collaborated with NetBeans around Java ME in a number of areas. Developers targeting Sony Ericsson devices can use the NetBeans Mobility Pack and the Sony Ericsson SDK for the Java Platform to create MIDP applications, as well as using Sony Ericsson's new Mobil JUnit testing utility. Additionally, developers targeting UIQ-based Sony Ericsson smartphones can use the NetBeans Mobility Pack for CDC to create Xlets and MIDlets for the CDC runtime.

TRIEMAX Software

TRIEMAX Software (*triemax.com*) is the manufacturer of Jalopy, a source code formatter for the Java programming language. It layouts any valid Java source code according to some widely configurable rules allowing projects to meet a certain coding style without putting a formatting burden on individual developers. Jalopy includes a plug-in for the NetBeans IDE and aims to support the widest range of coding styles and offers more than 400 distinct configuration options accessible through a graphical customization dialog, to let developers use their desired formatting output.

Conclusions

The above are just a taste of the many companies participating in the NetBeans Strategic Partner Program. To read more about the program, Strategic Partners, or apply for membership in either the Strategic or Community Partner Programs, please visit netbeans.org/community/partners. ☎



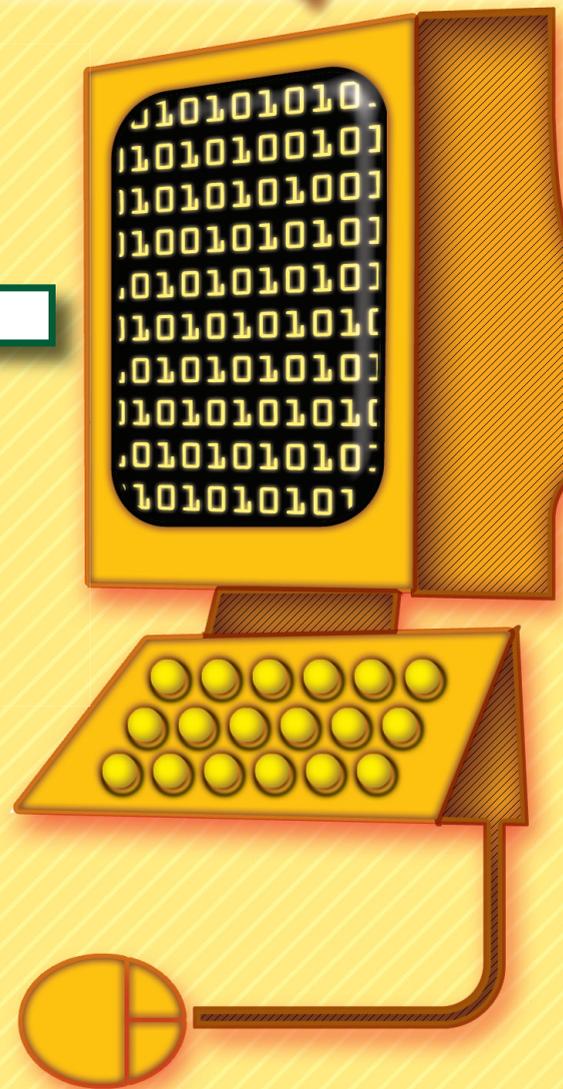
Matt Volpi (*matt.volpi@sun.com*) is the Product Line Manager for mobility tools at Sun Microsystems and is responsible for the NetBeans Mobility Pack.



BPEL

Designer

Kick Start



Exploring NetBeans' Visual Design Capabilities to Create a BPEL Process and a Composite Application

Sherry Barkodar

This tutorial illustrates deploying, executing and testing a synchronous BPEL process using NetBeans 5.5 with Enterprise Pack. A synchronous BPEL process represents a simple synchronous flow: it takes an input message and sends that message back synchronously. A client starts the synchronous process by invoking a request-response operation. After invocation, the client is blocked until the process finishes and returns the result.

Configuring the environment

Before you can deploy your application, the Sun Java System Application Server and JBI runtime must be configured correctly and running. To configure the environment:

1. Click the *Runtime* tab to open the *Runtime* window, and expand the *Servers* node.
2. If you have a default Enterprise Pack installation the Servers node already contains a *Sun Java System Application Server 9* node. In the *Runtime* window, right-click this node and choose *Start*. If the *Start* option is not available, the server is already running and you can skip the next step.
3. Wait until a completion message appears in the *Output* window (see **Figure 1**). When the server is running, the IDE displays a green arrow on the *Sun Java System Application Server 9* node.

Creating the sample project

From the IDE's main menu, choose *File>New Project*. In the Categories list, select *Samples>Service Oriented Architecture*, and then *Synchronous BPEL Process* (see **Figure 2**).

Click *Next*. In the Project Name field, type "SynchronousSample" (this is the default value). Optionally in the Project Location field, use *Browse* to select a different folder for the project files.

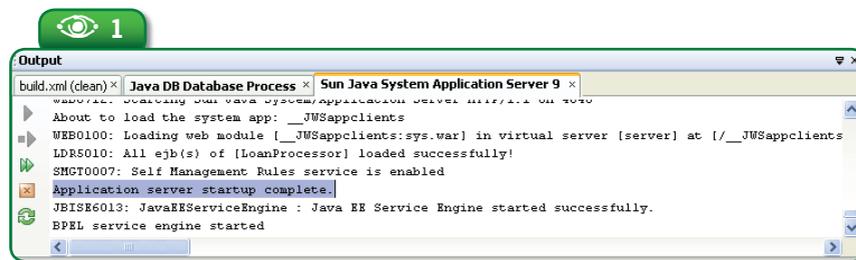


Figure 1. A successful server startup.

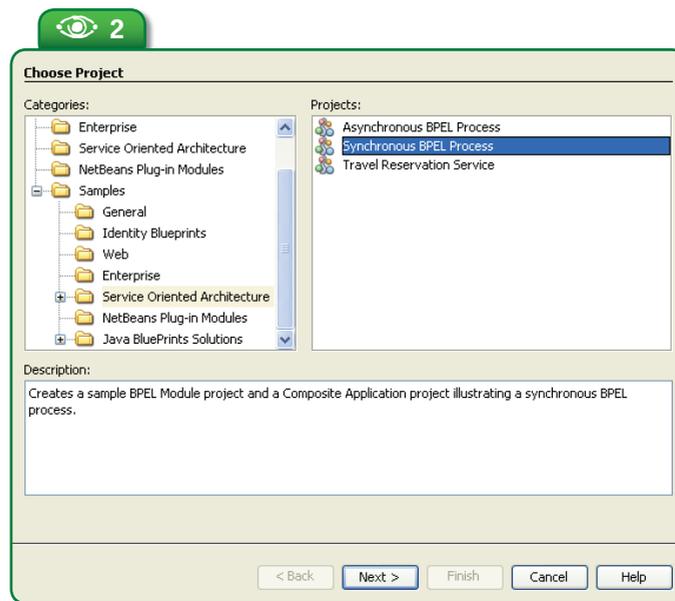


Figure 2. Creating the BPEL process

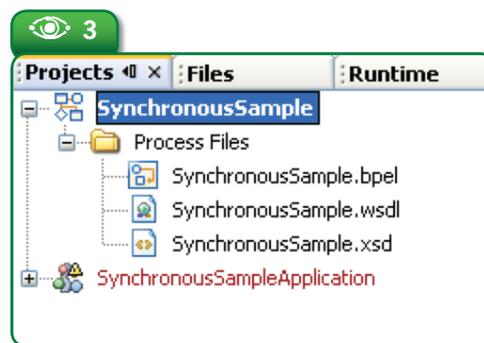


Figure 3. Files in the sample application

Click *Finish*. The Projects window now contains two project nodes: one for a BPEL project called *SynchronousSample*, and one for a Composite Application project called *SynchronousSampleApplication*.

Note that there is a broken reference in the *SynchronousSampleApplication* node. This is expected. The Composite Application has a dependency on the JBI Module. We'll resolve the broken reference later in the article, when we deploy the project.

Exploring the BPEL Project

In the *Projects* window, expand the *SynchronousSample* node; then expand *Process Files*. This node contains the following items (see **Figure 3**):

- *SynchronousSample.bpel*, the BPEL process
- *SynchronousSample.wsdl*, the process web service interface.
- *SynchronousSample.xsd*, the schema file.

Double-click the *SynchronousSample.bpel* node (see **Figure 4**). Notice the following:

- The Source Editor has a tab for *SynchronousSample.bpel*.
- The *SynchronousSample* diagram is shown in the Design view. The Design view allows you to visually model a business process. The BPEL Design tool automatically generates BPEL code that corresponds to the visual design.

▪ The Palette of BPEL elements is located to the right of the Source Editor.

▪ The Navigator window shows the BPEL logical view of the BPEL process.

Exploring the WSDL Editor

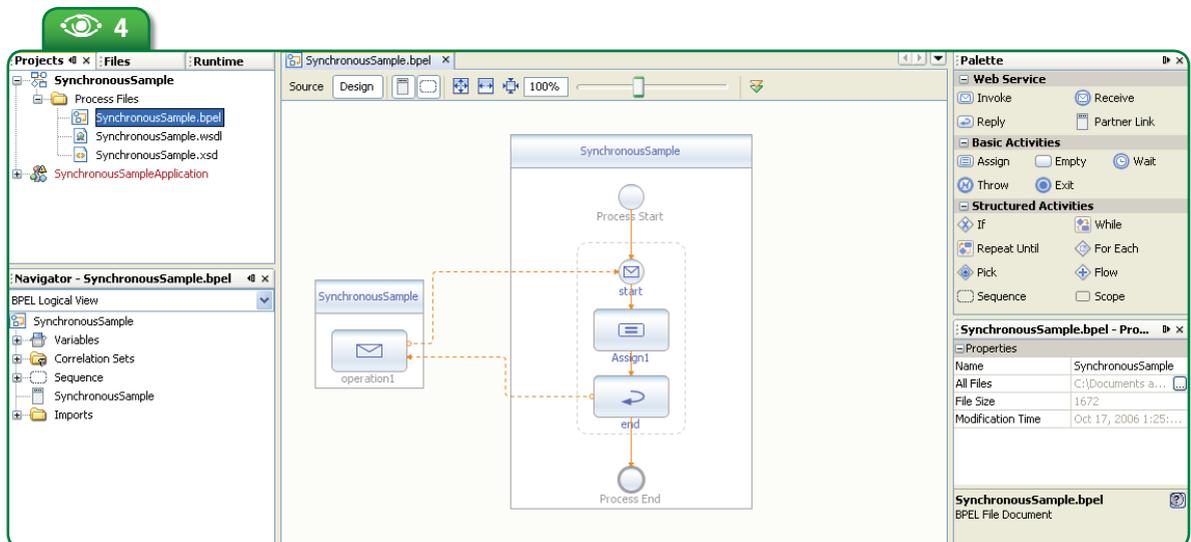
The WSDL Editor enables you to create and edit Web Services Description Language (WSDL) files. It includes a Design view and a Source view. To see the Design View, in the *Projects* window double-click the *SynchronousSample.wsdl* file. In this view, the WSDL file appears as a tree component where you can configure the file's elements and attributes (see **Figure 5**).

To see the Source View, click the *Source* button. The underlying XML source code appears, as shown in **Figure 6**.

Adding an If activity

In the *Projects* window, double-click on *SynchronousSample.bpel*. The IDE displays the *SynchronousSample.bpel* diagram in the Design view. In the Source Editor, change to the Design view (if you're not there already). In the *Structured Activities* section

Figure 4.
Designing the
BPEL process



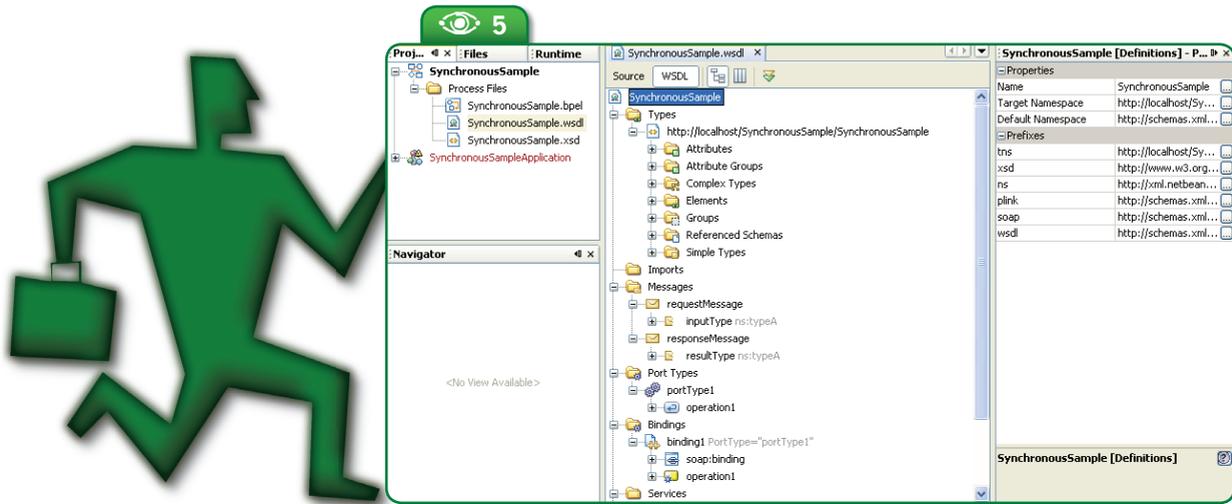


Figure 5.
WSDL Editor
Design view

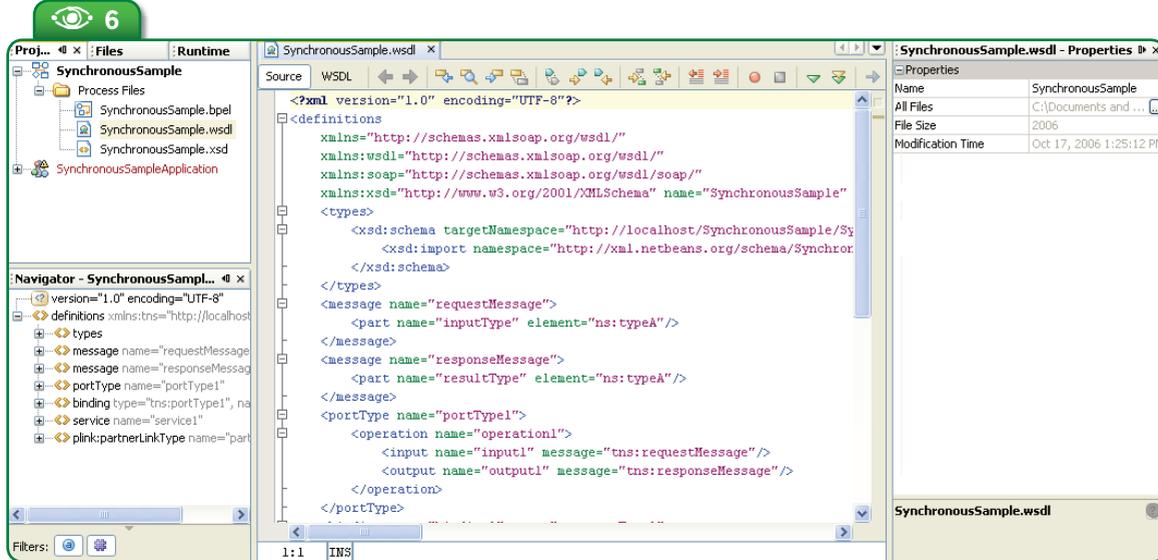


Figure 6.
WSDL Editor
Source view



Figure 7.
Adding an If
activity

Figure 11.
First Assign activity

of the Palette, select the *If* activity and drag it to the design area between *Start* and *Assign*. The IDE provides visual cues showing where you can drop the selection. This adds an *If* activity called *If1* (see **Figure 7**).

In the Design view, click the new *If1* activity. The BPEL Mapper appears at the bottom of the IDE (see **Figure 8**). (If the BPEL Mapper is not visible, choose *Window>BPEL Mapper*.) We'll use the BPEL Mapper to define the Boolean condition for the *If*. The sections at the top of the BPEL Mapper make up the Method Palette.

From the Method Palette, click *Operator* and select *Equal* (EQUAL). The *Equal* method appears in the middle portion of the BPEL Mapper (the Editor). Click *String* and select *String Literal* (String Literal) from the drop-down list. The *String Literal* box appears in the editor (you may need to drag it to the side). Type "Hello World" in the *String Literal* and press Enter. Then move the *String Literal* box and the *Equal* box to the middle of the editor (see the current mapping in **Figure 9**).

Figure 8.
BPEL Mapper window

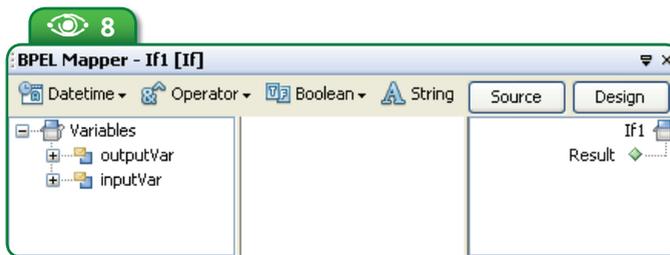


Figure 9.
First step of the mapping

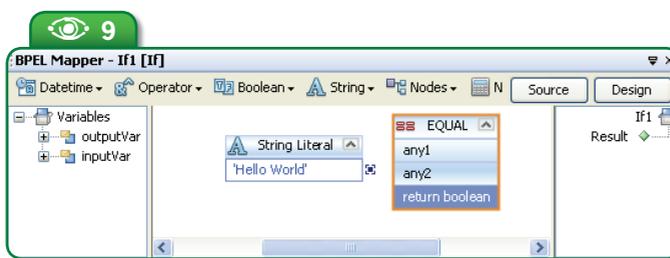
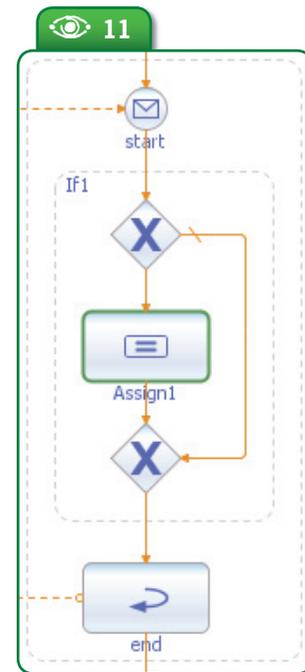
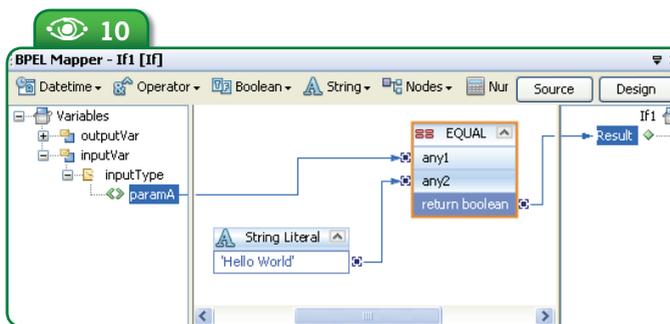


Figure 10.
Completed mapping



In the left side of the BPEL Mapper, under *Variables*, expand *inputVar>inputType*. Drag *paramA* onto the *any1* part of the *Equal* method. Select the small square on the right side of the *String Literal* box; then, when the hand cursor is visible, drag a connecting line to the *any2* part of the *Equal* method. Finally, from the *Equal* method, drag the *return boolean* section on to the *Result* (in the rightmost part of the BPEL Mapper).

Adding an Assign Activity

In the Design view, drag the existing *Assign1* activity to the *If1* activity. Place the activity between the two large X icons in the *If1* activity area, as in **Figure 11**. Open the *Basic Activities* section in the Palette and drag a new *Assign* activity to the design area, placing it to the right of the existing one (**Figure 12**).

Select the new *Assign2* activity. From the Method Palette of the Mapper, click on

String and select *concat* (Concat) from the drop-down list. The *concat* method appears in the editor. Double-click the first field in the *concat* box and type "Hello" followed by space, in the first string. Then press Enter or click outside the box.

In the left side of the Mapper, expand *inputVar>inputType*. Drag *paramA* onto the *string2* part of the *concat* method, the second field. On the right of the Mapper, expand *outputVar>resultType*. Drag the *return string* part from the *concat* method onto the *paramA* under *outputVar>resultType*. See **Figure 13**. This concatenates the string *Hello* to the input and copies the statement into the output.

Deploying the project

Before you deploy the BPEL project, you need to add the JBI module to the deployment project. Deploying the

project makes the service assembly available to the application server, thus allowing its service units to be run.

To add the JBI module, in the Projects window, right-click the *SynchronousSampleApplication* project node and choose *Add JBI Module* from the pop-up menu.

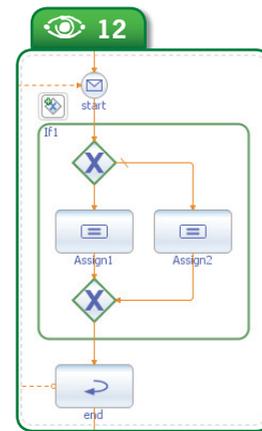


Figure 12. Adding a second Assign activity

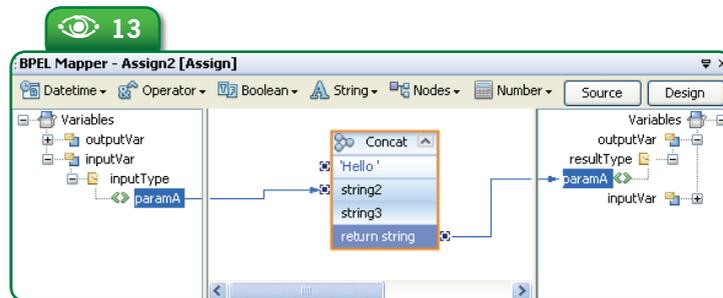


Figure 13. Assign2 activity mapping

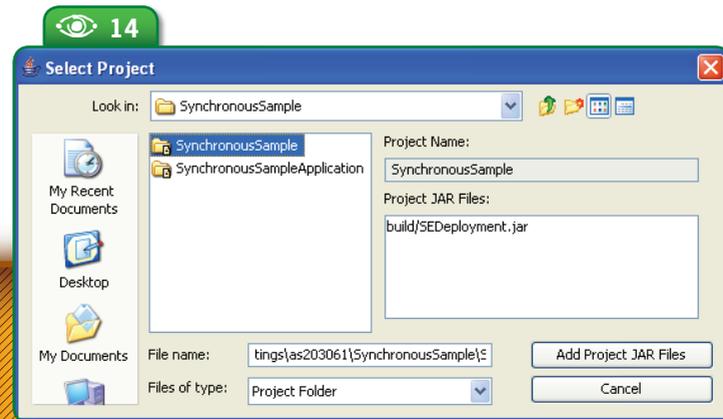


Figure 14. Adding a JBI Module to the Synchronous Sample Application

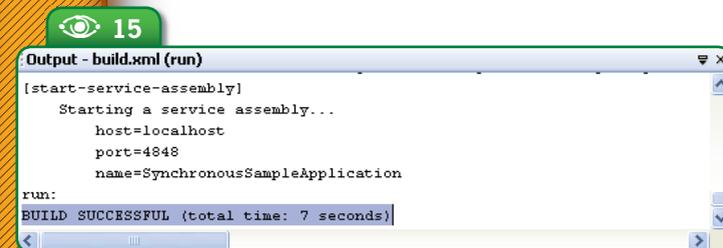


Figure 15. Deployment results for the Composite Application

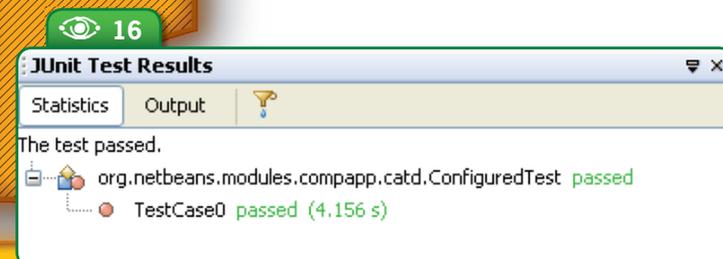


Figure 16. Results for the first test.



Figure 17. Confirmation for saving to an empty `Output.xml`

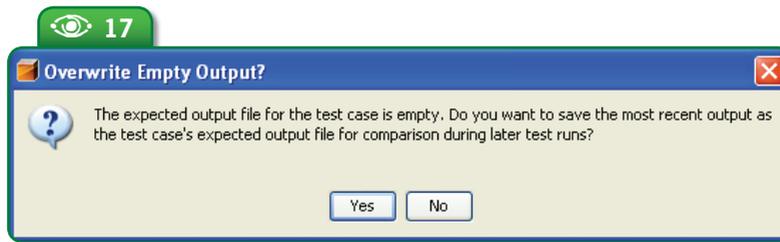
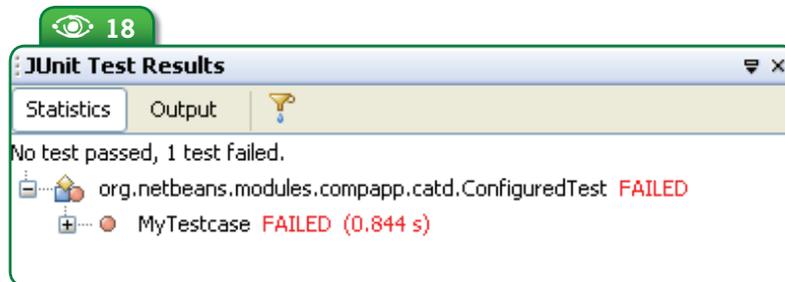


Figure 18. Results for the second test.



Select the *SynchronousSample* project and click *Add Project JAR Files* (see **Figure 14**). In the Projects window, expand the *SynchronousSampleApplication* project node and then the *JBI Modules* node. Notice that a *SynchronousSample.jar* node has been added.

To deploy the Composite Application, right-click the *SynchronousSampleApplication* project node and choose *Deploy Project*; then confirm the use of the SJSAS application server. You'll see a message like the one shown in **Figure 15** in the Output window (if the Output window is not visible, choose *Window>Output*).

Testing the composite application

You can enhance the Composite Application project by adding test cases, binding to the operation, supplying input, and then using the Tester.

In the Projects window, expand the *SynchronousSampleApplication* project node, right-click the *Test* node, and choose *New Test Case* from the context menu. The *New Test Case* wizard opens. For *Name*, enter (say) "MyTestcase" and click *Next*. Expand *SynchronousSample-Process Files*, select *SynchronousSample.wsdl* and click *Next*.

Select *operation1* and click *Finish*. Notice that in the project tree, under *Test*, a new folder *MyTestcase* is created, containing two files: *Input* and *Output*. Double-click *Input* and in the Body contents replace `<syn:paramA>?string?<syn:paramA>` with `<syn:`

`paramA>Sherry<syn:paramA>`.

Then save the project.

Double-click *Output.xml* and notice that before the test is run this file is empty. Each time the test is run, the current output is compared to the contents of *Output*. The *Input* will be copied to *Output* when *Output* is empty.

Testing the Synchronous SampleApplication

In the Projects window, expand to *SynchronousSampleApplication>*

Test>TestCase0. The *TestCase0* node contains two XML files: *Input.xml* and *Output.xml*. Right-click the *TestCase0* project node and choose *Run*. A pass message should be shown in the Output window (see **Figure 16**).

Right-click the *MyTestcase* node, and choose *Run* from the context menu. This is a special case where the *Output* file is empty, so the output is written to *Output*. Before that, a confirmation dialog is displayed (**Figure 17**); click *Yes*. Notice the failed message (**Figure 18**). Repeat the last test. After the first run, the *Output.xml* file is no longer empty, so its contents are preserved and are not overwritten by the new result. And the test passes.

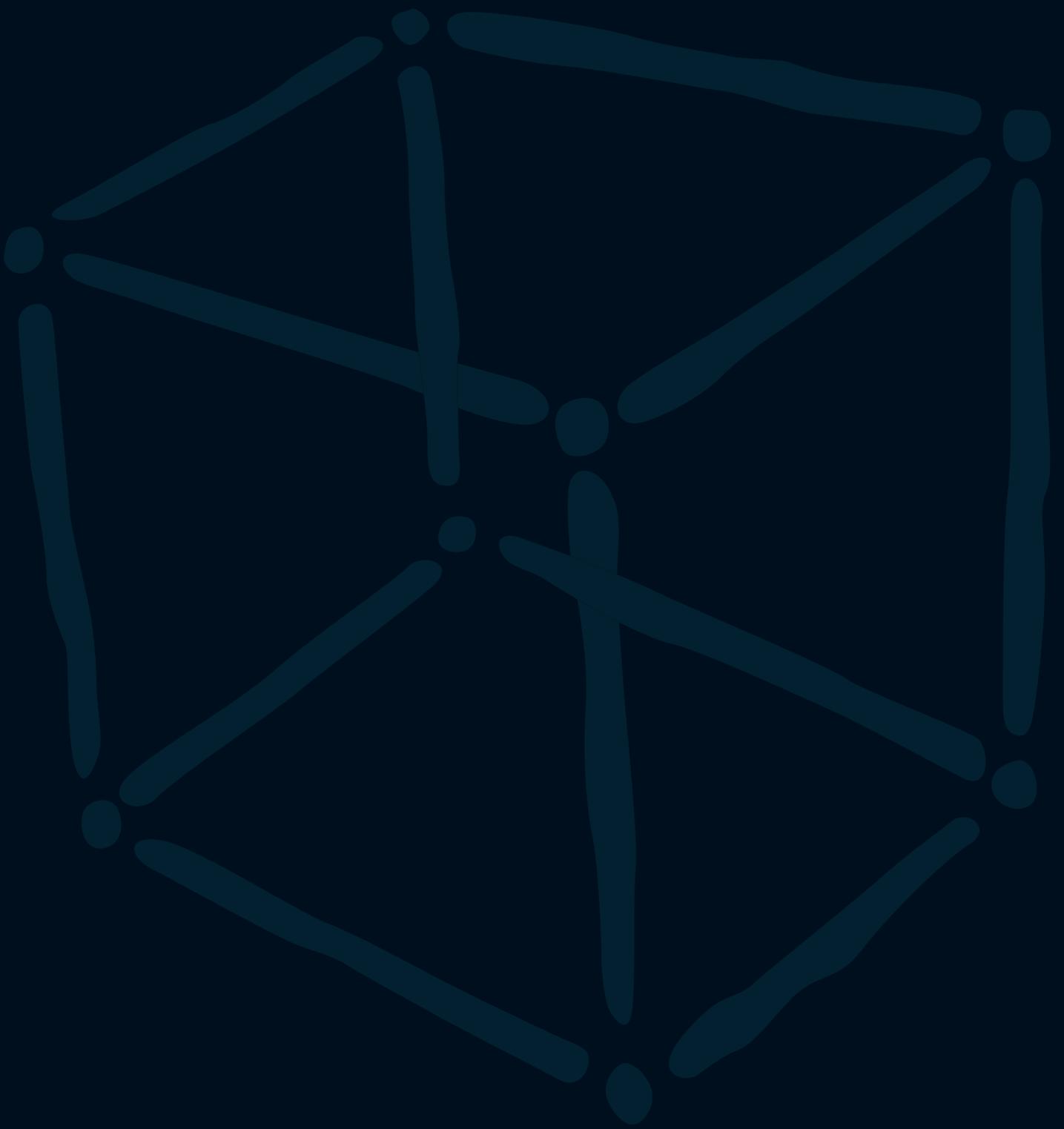
Conclusions

In this tutorial, you've seen how to use the NetBeans BPEL designer to create a synchronous process, and how productive developing, testing and deploying composite applications is with the new features in the NetBeans Enterprise Pack. 

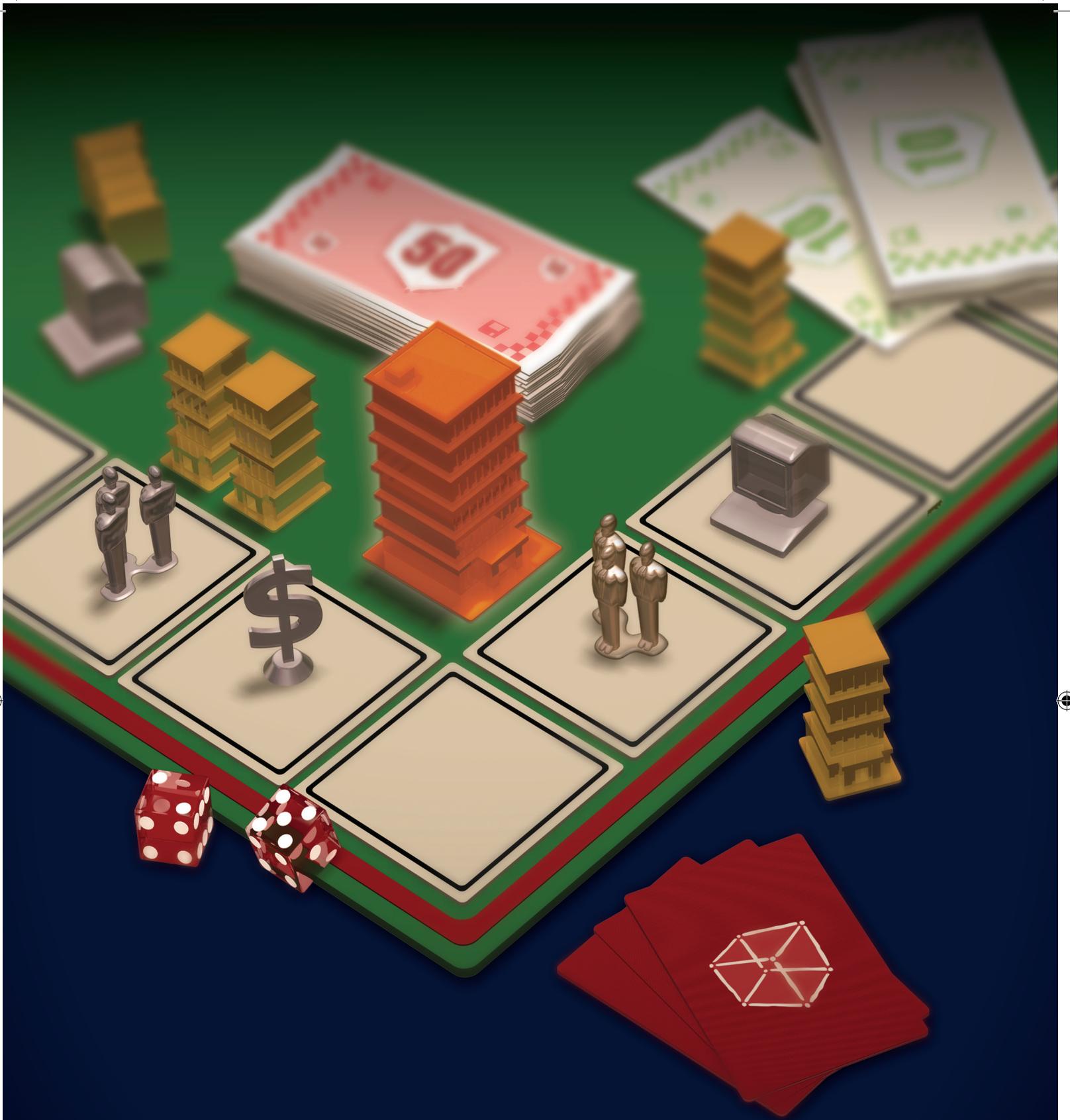


Sherry Barkodar

(sherry.barkodar@sun.com) has been with Sun for 7 years, and currently designs and implements Enterprise Pack tutorials and examples. She holds an architect degree from Tehran University and a MSCS degree from SJSU. You can visit her blog at blogs.sun.com/barkodar.



NetBeans



NetBeans