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
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 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 infrastructure, 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-
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:
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 the second one) 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 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!