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...
(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 Java-Doc 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.
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,
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).
Note that the “Namespace” combo box 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” compartment 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_,
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: stereotypes\(^2\). 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).

I 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.

**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
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-
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 other 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.
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
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...
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
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 `Syncronous 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()`, `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.