Improving existing Java code with a UML modeling environment

www.softeam.fr

www.modeliosoft.com

Abstract

The title of this white paper may appear somewhat surprising. How can a UML modeling environment be beneficial to an existing application where no prior model exists?

Despite the advantages of modeling an application before coding it, most Java developers are not using modeling tools for their developments. The “Improve your Java development efficiency with Modelio and UML" whitepaper demonstrates the advantages that a high-level modeling environment can bring. But what can be done when development has already been realized or begun without using modeling techniques? Are there still advantages to using a modeling tool?

We already know that modeling environments can generate Java code from a UML model. We have already published an article showing that a model-driven approach can bring significant gains in productivity and quality through the use of mature modeling tools providing high-performance code/model consistency management services (see "Improve your Java development efficiency with Modelio and UML"). This approach requires that models be designed and built first, before embarking upon Java programming.

And yet a new generation of tool is now freeing developers from this "constraint" (which is how it is seen by many Java developers). It is no longer necessary to first design and build a model, in order to take advantage of the productivity and quality gains that a modeling environment can bring.

This white paper will show how the Modelio modeling environment can enable you to improve existing code, enhance its documentation, and assist in the understanding of the architecture of a

Java application. These services constitute a first level of assistance, support and automation, which allow you to go even further with more elaborate use cases, such as the modernization of an application, the reverse documentation and reverse design of an existing application, the analysis of existing application architecture, and so on.

Modelio is a system and software modeling tool, which has been available in an open source version since October 2011 (www.modelio.org). Modelio provides a wide range of modeling features, such as the integrated support of all current modeling standards (UML, BPMN, SOA, ...), and the generation and reverse of Java code. A commercial version of Modelio, developed and published by Modeliosoft (www.modeliosoft.com), also exists, providing enterprise-dedicated solutions, with support services and additional tool features.

Reversing Java code in Modelio

Modelio has considerably enhanced its Java code reverse services, both in terms of efficiency and precision. From ".jars" or Java sources, Modelio automatically produces a UML model that precisely corresponds to the code in question. This model contains all the information needed to reproduce the code, with absolutely no losses or omissions: the UML static model (classes, packages, properties, …), the Java-specific extensions (stereotypes) and the textual code extensions (for example, method code) associated with the model elements concerned. From this model, exactly the same code can be generated.

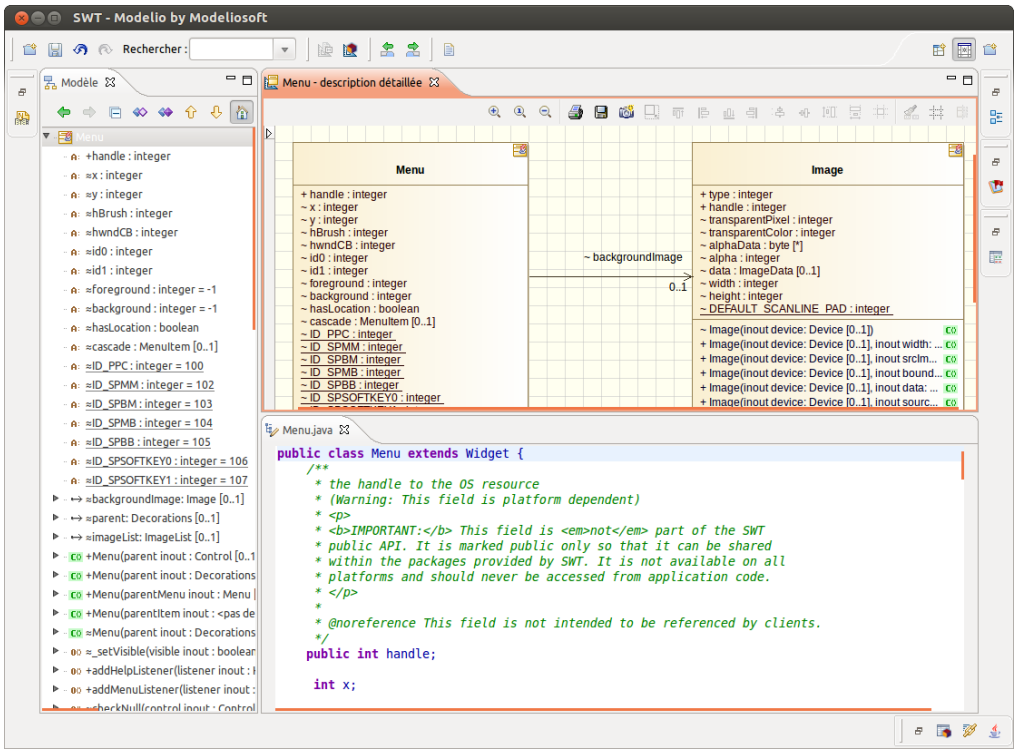

Figure 1 – UML model of a reversed Java application, with equivalent reproduced code

Modelio automatically builds UML static diagrams. The "*Create/Update automatic diagrams*" context menu command is used to tell Modelio to build the class or package diagrams from the

designated element. The models built enable you to focus on a class or package, and to summarize what is used and by whom.

At this point, you now have a model in which you can carry out all the operations useful to the Java code.

Producing improved Javadocs

Javadocs are very often used to document Java sources. Users of a Java library, for example, systematically use the associated Javadocs to find out which services are available, and how to use them. With Modelio (only in the Modeliosoft Java Solution), the "*Generate Javadoc*" context menu command lets you automatically produce the Javadocs associated with the model. These use the contents of standard Javadocs, and complete them by inserting the UML diagrams associated with documented elements. This improves the legibility of Javadocs, provides additional information (for example, who uses a given package), and facilitates browsing simply by clicking on elements in diagrams.

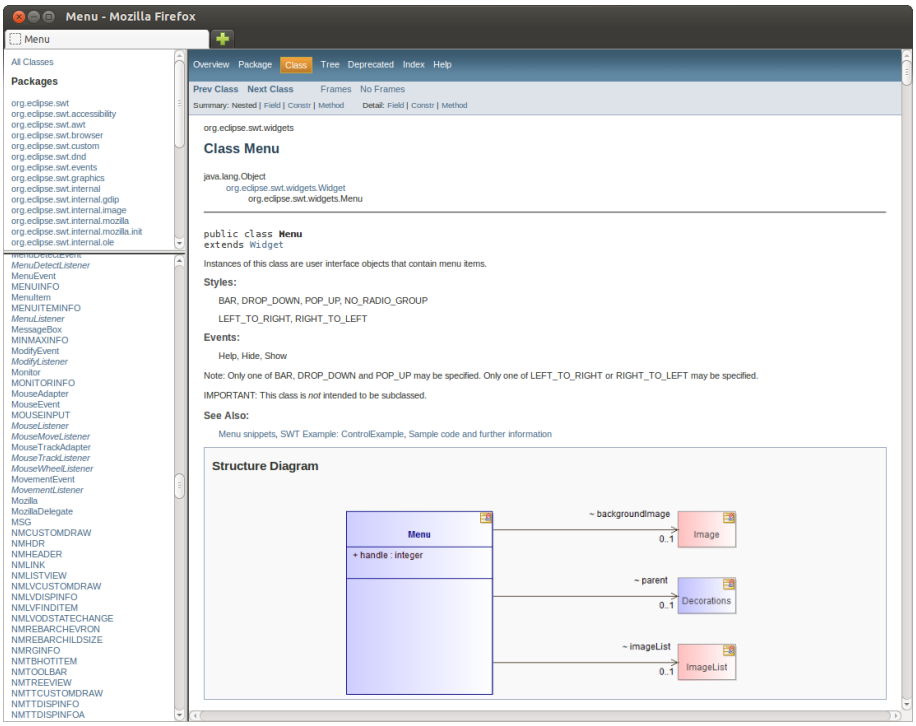

Figure 2 – Javadoc generated on the "Menu" class

Updating class utility methods

Java class utility methods, such as *toString()*, *equals()* and *hashCode()*, are commonplace and very frequently used by many of an application's classes. IDEs like Eclipse generate them automatically on demand. Eclipse produces code that is based on the attributes of the class. However, these attributes very often evolve during the development of an application, rendering the content of these

generated methods obsolete. This creates a large number of bugs in applications, as developers do not always remember to update this content.

Modelio (only in the Modeliosoft Java Solution) automatically generates these utility methods, and can maintain them when classes evolve, thereby guaranteeing permanent consistency.

In the case of Java code reverse, Modelio retrieves the code of these methods, and considers, by default, that it should not modify manual code. A dedicated macro is provided to extract the utility methods that are to be updated, and to proceed with regeneration.

Analyzing existing Java application architecture

Beyond the different types of dependency defined by the UML standard (for example "*import*, *access*, *package-import*, …"), the Modelio modeling environment carries out an analysis of all dependencies of all possibles kinds between classes, and creates a summary in the form of a kind of dependency named "blue link". This can be a Java "import" link, a class typing an operation parameter, an association, an inheritance link, an interface implementation, and so on. These dependencies are summarized at class and package levels. In this way, a dependency between packages in Modelio summarizes all the dependencies that exist between the classes of the origin package and the classes of the destination package. This provides a precise yet simple view of the Java architecture at package and summarized dependency levels, making it easier for architects to get to grips with the complex graph of dependencies that exist between the classes of a Java application. Packages are clearly materialized in Modelio, constituting a means of visualizing the overall structure of an existing application, and of restructuring it into layers that respect the usual rules of modularity and autonomy, while limiting circular dependencies. In Figure 3, 461 different dependencies between a large number of classes are summarized by 4 "blue links" between 5 different packages. Each "blue link" indicates what it summarizes, and Modelio enables users to examine summarized links in detail, where necessary.

This analysis, carried out here on packages, can be run at a finer level on Java classes, where incoming and outgoing dependencies to other classes are viewed.

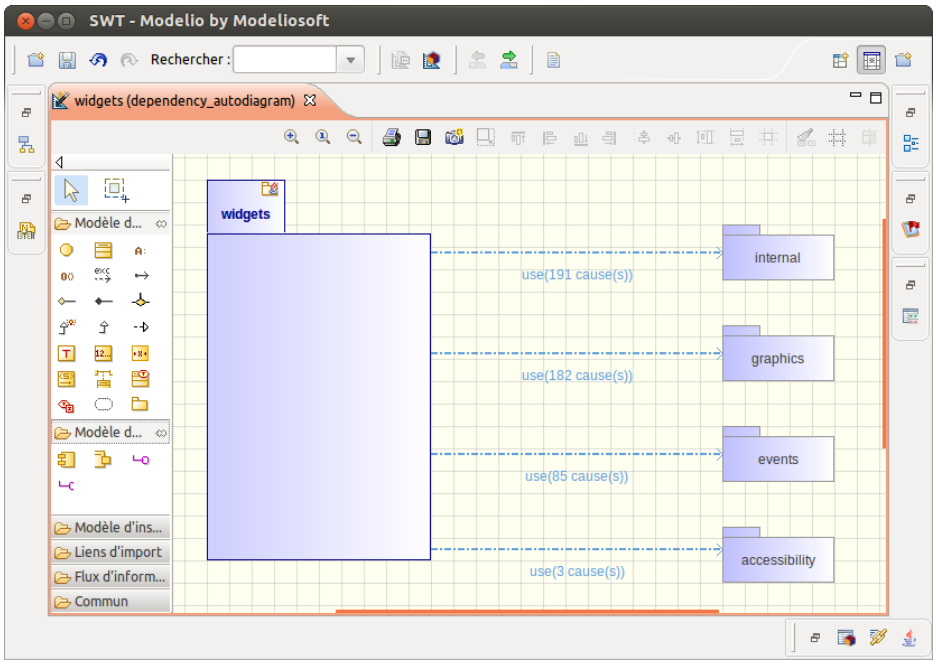

Figure 3 – Modelio architectural summary

Modelio links editor: Browsing the analyzed application

The Modelio links editor is a powerful tool, providing different views of an application according to what an architect wants to see. Options allow users to browse the application's inheritance links, interface implementation links, import links, "blue link" summary links, associations, or any combination of links desired. Figures 4 and 5 present two different browsable graphical views. The links editor presents the number of dependency levels chosen by the architect.

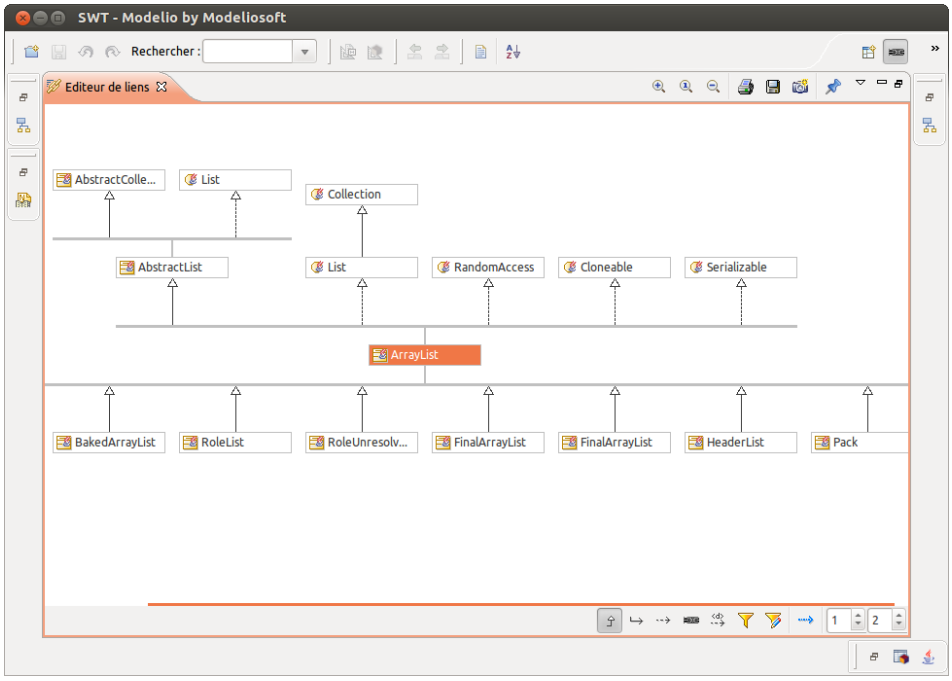

Figure 4 – Browsing inheritance links between classes

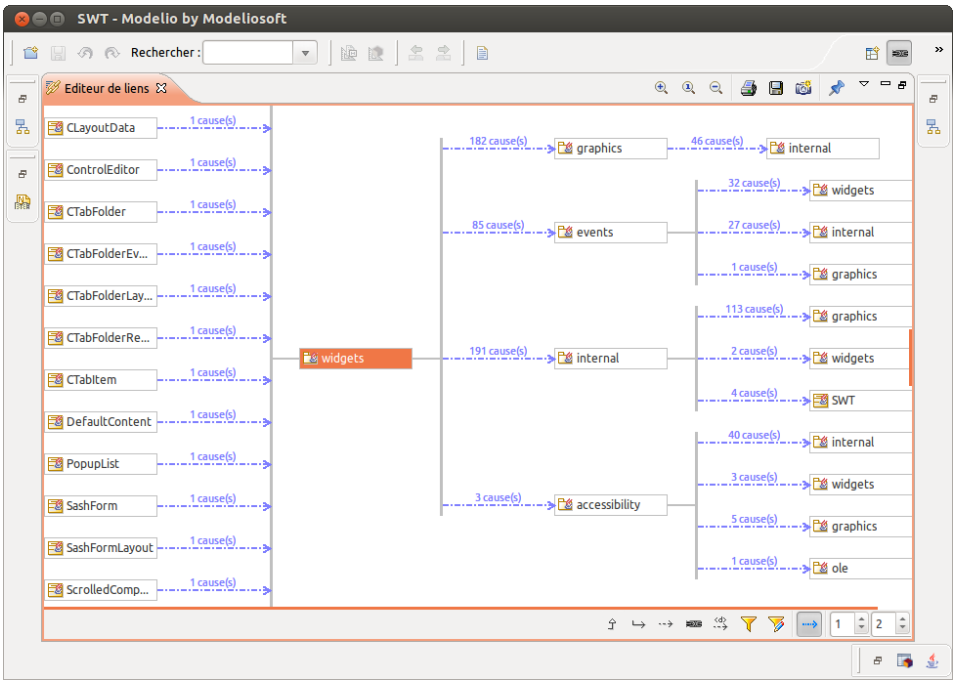

Figure 5 – Browsing (resumed) dependencies between packages

This browsing mode provides a much more operational and realistic view of complex applications than the simple hierarchical package/class view, making it easier to quickly understand an application. The links editor works in both directions (links originating from an element, and links arriving at an element). This means that you can carry out impact analyses on an existing application (What is impacted if such and such a class or package is modified? Who uses such and such a class or interface, and via which channel?).

Restructuring and modernizing an application

Once this Modelio-facilitated analysis has been completed, the architect can quickly reorganize an application. Firstly, he/she can easily modify the organization into packages in order to better structure classes, without impacting the final code and its execution. He/She can then regenerate an application and define a production in the form of reorganized libraries (.jar), and can also generate a production line (ANT, implementation of Maven, …) that reflects the new structure.

Should the architect want to operate in a more intrusive way, he/she can change the realization principles and underlying libraries, in order to obtain a more modern application. To do this, he/she can either use the "JPA" or "Hibernate Designer" Modelio modules, or create specific modules for a modernized architectural "framework". The "Pattern Designer" Modelio module can also be used to define and systematize new design patterns, which can be generalized across the application (see "Pattern Designer in action (3mn 45s)").

Finally, the development and modernization of the application can continue in "model-driven" development mode, even where the application was realized through direct Java programming.

Maintenance

The use of a model-driven approach helps combat the widespread syndrome of "architectural decay". Very often, the developers responsible for maintenance updates are not those who carried out the initial development of the application, nor those who designed the system. If these developers dive headlong into the code, they risk introducing sporadic corrections into different parts of the code in order to correct a bug, thereby altering and creating exceptions within architectural principles, and progressively ruining the entire architecture of the application. By starting out with the support of a model, everyone has an overview, which will help avoid this type of problem.

Conclusion

The volume of existing Java code to maintain and build on is considerable, and it quickly becomes extremely difficult for those in charge of maintenance to keep up in terms of their knowledge and their capacity to carry out global restructuring or correction operations. For this reason, help is needed from tools able to provide overviews, handle large volumes of code and update what already exists. The use of models has proven to be extremely useful to abstraction, comprehension and production.

With the approach provided by Modelio, models do not need to have been used since project start-up. The Modelio approach is applied to Java code throughout all development phases, thereby ensuring smooth communication between developers and architects. Modelio can be used to document, assist comprehension of and build upon and restructure what already exists.

Useful Links

- www.modelio.org: The Modelio open source community website. Modelio can be downloaded here.
- www.modeliosoft.com: The website of the original author of Modelio, where commercial solutions are distributed.
- http://www.modeliosoft.com/modelio-store.html: The Modelio Store, where Modelio modules (extensions) such as Java Designer can be downloaded.
- http://www.jcp.org/en/jsr/detail?id=901: Java language specification.
- http://www.omg.org/mda/specs.htm#MDAGuide: Model-driven technologies and standards.