Improve your Java development efficiency with Modelio and UML

Philippe Desfray – SOFTEAM, Modeliosoft

Philippe Vlaemynck - SOFTEAM, Modeliosoft

Copyright SOFTEAM 2011

www.softeam.fr

www.modeliosoft.com

Abstract

Modelio is a software and business modeling tool that has had an open source distribution since October 2011. It provides a number of interesting features, including support for all current modeling standards (UML, BPMN, SOA, ...) and Java code generation and reverse.

Despite the advantages of modeling an application before coding it, most Java developers are not using modeling tool for their developments. Modeling allows them to define the architecture and its evolution, and also to communicate this information efficiently to others in the development team. The problem is that once developers begin coding, ad-hoc changes to the code are not reflected in the model, and the model gradually loses sync with the code and becomes useless. This is the one-shot model syndrome…

This white paper will walk through typical Java modeling use cases and describe how Modelio can be used to model Java architecture. It will also demonstrate one of Modelio's most useful features - its ability to automatically maintain consistency between the code and the model, so that any changes made to the code will automatically update the model and vice-versa. Modelio is the only open source modeling toolkit to provide these features, thus ensuring the long-term utility of the model for Java developers.

This white paper will also discuss the benefits of this approach, explaining the tradeoffs and highlighting the functionalities which are most useful in the support of a balanced and efficient model/code development approach.

Understanding Developer Productivity

Most developers understand productivity as being about producing a lot of code as quickly as possible, because this is what they experience on a daily basis from management and customer pressure. Therefore, many developers think that UML modeling will not increase their productivity by much. They're not completely wrong: drawing a class diagram and filling in attributes and operation signatures is no faster than typing the same information into a class skeleton. Consider the model in Figure 1 and its corresponding Java code, and you'll see that there is not much evidence of enhanced productivity, especially if the Java developer is already using an IDE such as Eclipse.

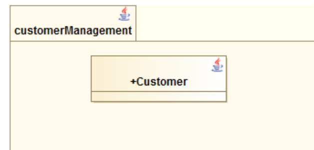

Figure 1 – The simplest model of a Class in UML

```
package customerManagement;

import com.modeliosoft.modelio.javadesigner.annotations.objid;

public class Customer {

}
```

The Modelio Java Designer module can produce more sophisticated code depending on the model, for example, associations, as shown in Figure 2, and useful additional Java code, such as getters and setters requested for the association:

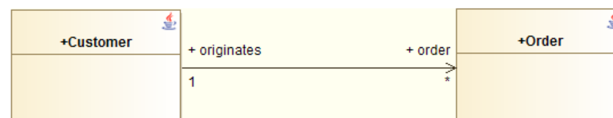

Figure 2 – An association between two classes

```
package customerManagement;

import java.util.ArrayList;
import java.util.List;
import com.modeliosoft.modelio.javadesigner.annotations.objid;

public class Customer {
        public List<Order> order = new ArrayList<Order> ();


    public List<Order> getOrder() {
        // programmers code to be entered
        return this.order;
    }

    public void setOrder(final List<Order> value) {
        // programmers code to be entered
        this.order = value;
    }

}
```

© SOFTEAM 2011

However, even with the above example, one might argue that any advanced IDE (like Eclipse) can generate such getters and setters. So why should a developer bother with creating a model and using a modeler when a simple IDE can carry out this kind of automation?

The value of the modeling approach can be seen when the previous example is extended forward in time. Imagine that, after a few iterations of the code, the association end is dragged and dropped to another class, or that the destination class name is changed (Figure 3). The code generated by the IDE is now obsolete and will remain so until the developer requests the IDE to re-generate new accessors for what appears to be a new attribute in the class. The developer will also have to clean up the code by removing the obsolete accessors, as this is not done by the IDE. Now, the developer will quickly realize that the automation provided by the IDE was actually only a one-shot exercise.

When the developer is using a modeler, the regenerated code is updated accordingly. No risk of forgetting anything, no mistakes, and no trouble.

Figure 3 – The association destination has been changed (command)

```
package customerManagement;

import java.util.ArrayList;
import java.util.List;
import com.modeliosoft.modelio.javadesigner.annotations.objid;

public class Customer {
    public List<Command> command = new ArrayList<Command> ();


    public List<Command> getCommand() {
         // programmers code to be entered
         return this.command;
    }

    public void setCommand(final List<Command> value) {
         // programmers code to be entered
         this.command = value;
    }

}
```

This small example shows that productivity should be considered not only for the initial "first run" use case, but also for real working situations where changes to the code are frequent. As any developer is aware, it is certainly much more tedious to update code with incremental changes, than to create code initially. However, in practice, these changes happen in every code tree that has a large number of classes. Without a proper modeling tool, it is difficult to make sure that everything is in sync and properly updated.

More generally, code writing is a small part (30% at most) of the software development effort, which includes analysis, design, testing and validating tasks. And this development effort is itself only a part of the global maintenance effort, which is frequently evaluated as costing 15% of the initial development effort per year. These figures have to be considered for well-architected software. The situation is obviously much worse for poorly-architected software.

© SOFTEAM 2011

The Power of Modeling Abstraction

UML modeling brings its best benefits during the analysis and design phases.

Communication Support

Analysis, use cases, global architecture, and patterns are well supported by UML. Models provide an overview of problems and enable communication between analysts, designers, developers and any other stakeholders (at their level of interest). Models also provide additional semantics that do not exist in programming languages. As a small example, associations carry semantics that are lost in the Java code, such as cardinalities, composition (if any), associated constraints, information on the opposite (non-oriented) association end, and more. You can understand architecture by looking at a picture (a class or package diagram) at the UML level, whereas you may need to scrutinize thousands of lines of code to get the same information from Java code.

Abstraction

Using the abstraction power of models, a single UML class can offer much denser information than a Java class. For example, let's consider a conceptual model of the entities to be managed by an application. Such a model will be of significant value to analysts, designers and developers, since it provides a common understanding of the underlying concepts of the application. In addition, depending on the technical choices and constraints of the project, such as storage in a relational database, a related GUI, and the application of several framework-specific patterns (e.g. J2EE, Spring, ...) on it, one UML class of the model may be translated into a large number of Java classes.

Reduction of Complexity

In practice, the benefits of using models increase with the size and complexity of the software. Developing a software product containing 100 classes is not always a big issue, but designing an application that uses 10,000 classes requires consistent software architecture and design throughout the entire development cycle.

Architecture is the key quality factor for Java developed code. Developers know that well-architected applications are easy to program. For each change in functionality, there is at most one place where the code has to be modified, and this effect can be seen across the entire application. Modeling helps developers to concentrate on the overall picture, and to define architecture that is disconnected from low-level code. Models can be used to master architecture, to drive development and to facilitate evolutions and changes. Of course, models must always be kept in sync in order to remain useful.

Maintenance

Following a model-driven approach helps fight the "architecture decay" syndrome, where developers performing maintenance updates are frequently not the developers who originally performed the application development, nor the architects who designed the system. If they dive too quickly into the code, they may "hook" the code from place to place to fix bugs or introduce evolutions and exceptions to the architecture principles, progressively disturbing the entire architecture of the application. Starting from the model gives them a global view and helps prevent this occurrence.

© SOFTEAM 2011

Best Practices

The model can also be a guide for less experienced developers. Even when used as a drawing aid, it invites developers to think before they act, and to define the model before diving into programming. Furthermore, model diagrams can be partial and show only certain elements, keeping a particular concept or design point clear of other secondary concerns. This ability to focus on specific aspects helps developers to better understand a design, while code generation tools provide them with a framework to help them create standards-compliant code.

Modelio Technologies for Model/Code Synchronization

To gain these benefits, code/model consistency is key. None of these benefits are possible if there isn't a smooth and robust code/model synchronization process in the modeling tool. If Java developers find that they have to do the job twice – once at the model level and again at the code level – as well as manually making sure that both levels are in sync, they will leave aside the modeling work, and go straight to the code with their favorite IDE. At best, there will remain an obsolete and dusty initial model that reflects only the initial design intention and nothing else.

That's where Modelio comes in.

Modelio is based on the architecture of Objecteering, a modeling tool at the cutting edge of innovation and model-driven development for 20 years. Objecteering has always been well-known for its model-driven developmentn first with C++ from 1991 onwards and then with Java since 1996. Modelio was recently been released under the GPL v2 open source license, and provides a complete open source modeling product, with support for both the UML and BPMN standards.

In addition, the new open source Modelio project has been given a modular architecture, with the key APIs (in Java) licensed under the more permissive Apache 2.0 open source license. This allows individual developers, communities and partner companies to develop both open source and proprietary add-on modules, providing new specialist functionality to the core product.

Modelio provides a roundtrip mechanism that minimizes the consistency maintenance effort, by providing a close integration with the IDE (Eclipse) and by allowing changes in the code within operations (algorithms) or within structure definitions (classes, attributes, generalizations, ...). Modelio keeps the abstraction level unchanged. The model is not rebuilt from the code, but is rather updated by code changes. For example, associations are maintained even after a roundtrip session.

Let's take a closer look at how Modelio manages code/model synchronization.

© SOFTEAM 2011

Three Levels of Modeling

Depending on the phase of the development cycle (analysis, design, or detailed design/coding), the model parts implemented are not identical.

- During the analysis phase, models are used to build the application's dictionary, define its requirements, realize use cases, construct design models and use all of UML's modeling capacities. Model construction and documentation generation are the main results of this phase.
- The design phase focuses on realization, by building productive models that concentrate on the main architectural themes and essential system classes. Designers are encouraged by the fact that their models will have an immediate translation into Java code in the programming phase.
- The coding/detailed design phase completes the general design model in roundtrip mode, by using the features of the Eclipse/Java environment and by permanently synchronizing the model.

Model-Driven and Roundtrip Code Generation

Figure 4 shows Modelio with one UML window in sync with one Java window. Most frequently, developers capture their Java code in Eclipse.

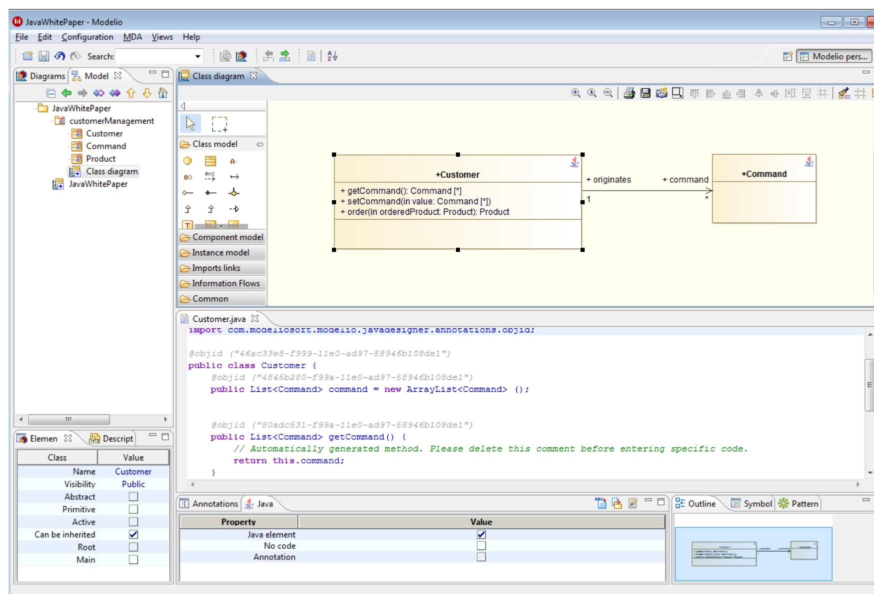

Figure 4 – Modelio Java Designer permanently manages UML model/Java code consistency

The model-driven approach must balance the advantages and disadvantages of model orientation or code orientation. Modeling everything before coding can be excessive in the case of small technical Java classes. However, a code-centric approach can result in the production of huge amounts of often badly constructed code.

For example, Eclipse is a great tool for productivity, with code "refactoring" features and handy completion mechanisms. However, completion (for example, when

© SOFTEAM 2011

Model-Driven Generation

With this approach, the model is developed and then the code generated, with authorization to write method code only between markers in the Java code. Modelio is therefore in charge, with Eclipse being used to complete the skeleton of produced code. In theory, this is the recommended mode, since it guarantees that model-level design is respected. Below, we see the comments marking up the beginning and the end of code that can be changed by the programmer (procedural code).

```
public Product order(final String orderedProduct) {
//begin of modifiable zone(JavaCode)......C/3a0ca6e8-f99c-11e0-ad97-58946b108de1

. . .you can enter your application code between the markers

//end of modifiable zone(JavaCode)........E/3a0ca6e8-f99c-11e0-ad97-58946b108de1
//begin of modifiable zone(JavaReturned)..C/3a0ca6e8-f99c-11e0-ad97-58946b108de1

//end of modifiable zone(JavaReturned)....E/3a0ca6e8-f99c-11e0-ad97-58946b108de1
    }
```

Note that in model-driven mode, the developer is not expected to introduce any additional code outside the markers, i.e. he or she cannot add any attribute or method definitions, imports or classes, nor can he or she modify any existing method signatures. This makes the model-driven approach a highly secure, if somewhat constrained, mode.

If this mode is well adapted to design phases when code is not yet the main concern, it is also valuable during maintenance phases, as it guarantees that changes in the code will not result in any unwanted design consequences. However, this mode is usually too constraining for the development phases where some agility is required due to the numerous refactoring activities that usually occur during such phases.

Roundtrip Generation

With this approach, the code can be freely altered. Modelio then resynchronizes the model with the code by reversing it. In this case, Eclipse is in charge, with Modelio adapting the model to the code (Java reverse). This mode is popular among developers, since they are free to take full advantage of the power of Eclipse/Java. We see below from the code that only one marker is generated to retrieve the operation by its identifier.

```
@objid ("3a0ca6e8-f99c-11e0-ad97-58946b108de1")
    public Product rder(final String orderedProduct) {
    }
```

The roundtrip mode is obviously more flexible than the model-driven one. However, as more freedom is granted to developers, special care must be taken to ensure that the overall design is not damaged by ad-hoc modifications. In such situations, model reviews after reverse operations are important and worth considering.

Modes and Development Phases

The strict model-driven mode is typically used during the design phase, which does not go into detail concerning highly specific Java classes, such as in the case of GUI construction. For example, an "ActionListener" class will not be modeled. The GUI is simply built using Eclipse features, possibly using specific complementary tools. However, the main dialog classes are modeled and then generated, along with the generic GUI classes, and the rest is carried out in Eclipse and then reversed

© SOFTEAM 2011

in Modelio (using the "Update from source" command). The implementation phase uses the roundtrip mode, which is more flexible for the programmer, while keeping design classes in model-driven mode.

Developers need to understand the benefits that they get from using modeling techniques. Using Modelio and UML to help with Java development still leaves all the responsibility on the shoulders of the developers. In roundtrip mode, they have the freedom to change any parts of code they want, but they have to know the model and respect the design intention, in order not to provoke any architectural decay. Giving developers this responsibility is the best way of getting them on board. In practice, developers using Modelio for Java development generally have two screens linked to their workstation - one screen dedicated to development (Eclipse-Java), and the other used for modeling (Modelio).

Other Useful Features

ANT File Generation

Modelio Java Designer also generates ANT files, and compiles and produces executable code for Java, based on models similar to the one shown in Figure 5. This is useful for complex applications where an application/package mapping has to be modeled in order to produce the targeted binaries. The production scheme is thus modeled, and simply drives the generation code.

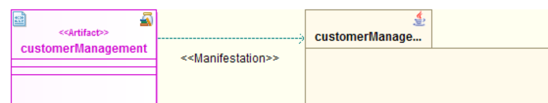

Figure 5 – The Artifact represents the library or binary produced from the "manifested" packages or classes

Pattern Automation

Design Patterns can be modeled within Modelio, simply by defining models and pattern parameters (Figure 6). Pattern application will then create instance models according to actual parameters, and produce the corresponding Java code through the generation process. This automates systematic pattern development, a frequent task when using frameworks or libraries.

© SOFTEAM 2011

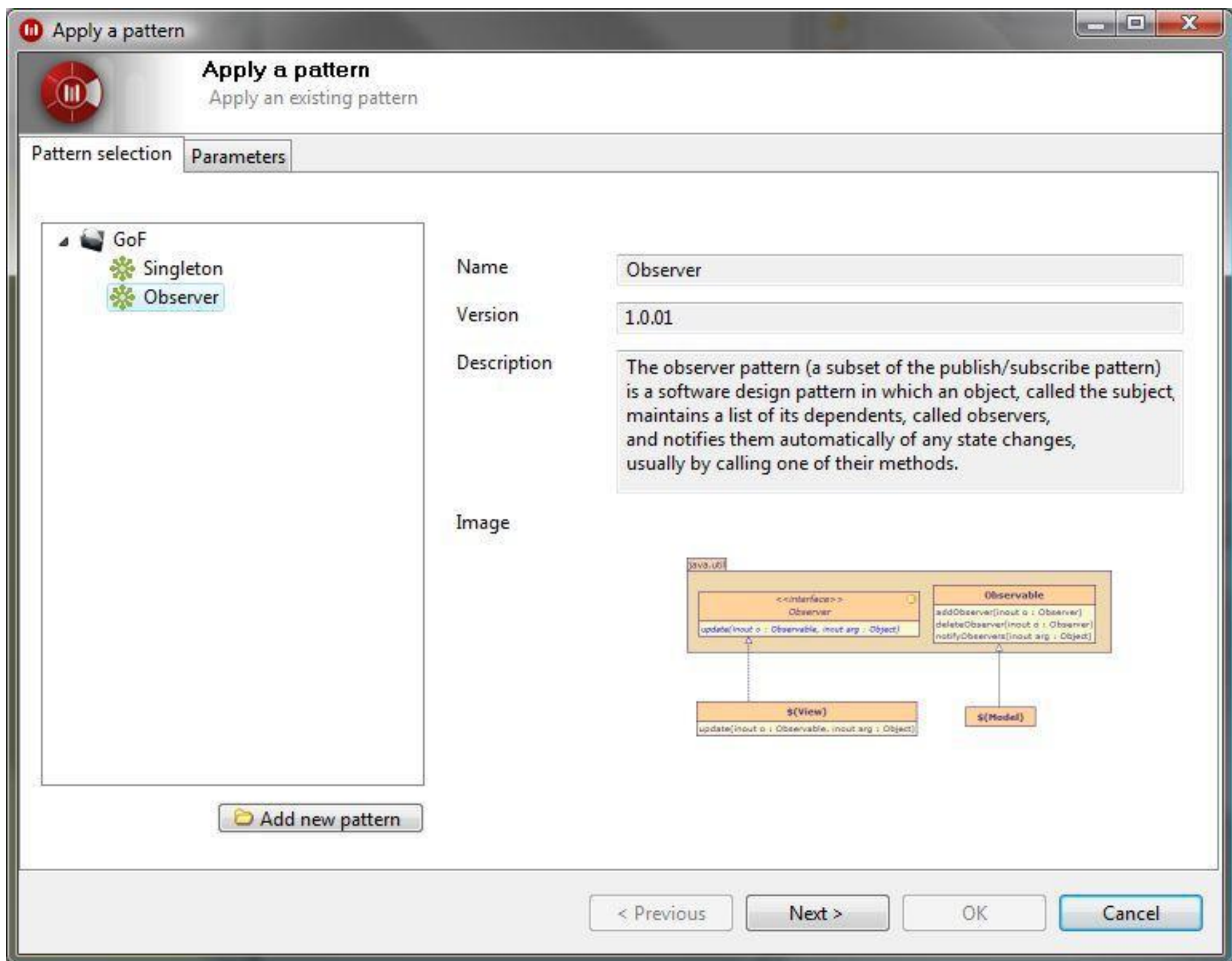

Figure 6 – Modelio Pattern definition wizard

Profiles

Modelio can be extended by defining profiles. A profile helps to map a model to a specific target or framework. As an example, stereotypes defined in a profile can be used to specify that a class is "persistent", that an attribute is an "identifier", and so on.

Open API and Metamodel

Modelio provides a rich Java API and an open metamodel, that allows automation of any additional code generation or model exploitation tasks. Advanced examples such as generating code from state machines or mapping to frameworks can use this feature.

Community Modules

The modelio.org website provides "modules" (Modelio packaged extensions) for different automation targets, such as Hibernate mapping. Thanks to the fact that Modelio is open source, programmers can build their own modules for any modeling automation or code generation tasks, and combine them with the existing Java Designer module.

© SOFTEAM 2011

Conclusion

Productivity is more than simply generating lots of lines of code from a simple model. It can only be measured at the scale of a team working on a project throughout the entire development cycle.

A modeling tool is not a silver bullet. If you use a bulldozer without being able to drive it properly, you're likely to end up doing more damage than good. However, there are genuine advantages of following a model-driven development approach and these make it worthwhile to put in the effort involved in setting up a proper model development approach within a team.

There is certainly a resistance to change that prevents developers from using a model-driven approach. Expensive modeling tools that may not smoothly maintain code/model consistency throughout the development phase are a significant hurdle. Now, with Modelio Java Designer becoming open source, it is easier to improve development productivity and quality through model based tooling.

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, who distributes commercial solutions here.
- 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.

© SOFTEAM 2011