Ubiquitous MDSD – Bringing Together Modeling and Coding

Model-Driven Software Development (MDSD) is a powerful paradigm that has finally reached mainstream software development in the course of this decade. But although we have a large toolset at hand for both modeling and coding – most notably provided by the Eclipse EMP and JDT projects – the integration of the two worlds is often rather bumpy. This paper shows an approach that bundles graphical model editors based on GMF and code generators based on openArchitectureWare (oAW) into a regular Eclipse IDE and strives to achieve a seamless transition from modeling to coding (and back) in everyday development.

Motivation

Over the years, integrated development environments (IDEs) have boosted developer productivity in many ways. We have grown accustomed to features like content assist and automatic code completion, fast and easy navigation through code, automatic and incremental code compilation, and on-the-spot error indication, to name but a few. All this has become possible because we have been doing the same thing for a couple of decades now: writing source code.

Enter model-driven software development. We identify recurring patterns in our software, find formal abstractions to specify instances of these patterns (called metamodels, then), and tell our developers to create models instead of source code, because we can generate source and other artifacts out of these models very easily.

And while it is hard to deny the increases in productivity and quality gained by generating code out of formal models, many developers find working with models very cumbersome and annoying. This is very often caused by the lack of productivity features like the ones mentioned above in MDSD tooling. Another common burden is that now developers must live in two parallel worlds: the model and the code, interconnected solely by the task of running the code generator.

It is thus our primary goal to make the model driven software development process as productive as the source-code-only software development process already is, and to provide a seamless integration of modeling and coding, up to the point where the developer no longer feels any difference between the two of them, thus making MDSD truly ubiquitous.

This paper first puts the solution into perspective by presenting the framework into which everything is embedded. We then pick out some specific areas where we have taken steps towards reaching the goal of ubiquitous MDSD, and conclude with an outlook of topics yet on our roadmap.

The Framework

Our main target is to simplify the development of business applications, so we focused on a rather abstract, technical domain of (persistent) business objects and their interrelations. A standard, GMF-based graphical editor was developed in a rather short time-frame, while at the same time we implemented a flexible, aspect-oriented framework for managing the business objects at runtime. We had a similar flexibility in mind with regard to code generation, and we chose the openArchitectureWare (oAW) generator framework as a technical foundation.

With the goal of ubiquitous MDSD in mind, it is imperative to hide from the developer as many of the tooling details as possible. Just as a Java developer typically is not interested in the fact that Eclipse uses its own Java compiler, it is of no importance to know whether the model editor is GMF-based or that the oAW generator interprets a set of templates in order to create the desired code. So while we are using all of these technologies, their existence is largely hidden from the developer. This is the arena where we can begin working on the goal of seamless integration.

Tool Integration

First and foremost, the tool integration itself is a primary concern. It is most important that all tools required by the developer are integrated into one platform. Switching back and forth between a modeling tool and a separate IDE, possibly through the interruptive step of a manual generator invocation in a console, is most unproductive and by no means satisfying for the developer. The Eclipse platform obviously presents the optimal integration hub to achieve this goal.

In order to hide generator details from the developer, we need to encapsulate raw generator templates into modules or cartridges and bundle them as Eclipse plug-ins. These bundled generator cartridges make themselves known by registering an extension at a central generator component. The developer can choose which cartridges to use, and of course define the model they are supposed to operate on. The generator can then be run in a pre-defined workflow using this configuration information.

The result is that in his IDE the developer finds ready-to-use bundles, which are capable of generating code (or any other textual artifact), and which he can select to operate on his model. Unless he wants to contribute his own generator cartridge, all details about openArchitectureWare, metamodel configuration, template syntax, and workflow descriptions are hidden from the developer.

Error Detection and Indication

With MDSD, it has often been a problem for developers that errors in the model are only detected when running the code generator. Even then, cryptic error messages, exceptions, or simply mal-generated code were often the only result.

It is therefore important to detect and indicate errors as soon as possible, i.e. typically when creating and saving the model in the graphical editor. GMF provides an excellent integration with the EMF validation framework. (OCL-)Constraints can be formulated in the metamodel and are checked more or less automatically. Error markers are shown both within the diagram and in Eclipse's problem view – the very same place where Java compiler errors will show up. This may seem trivial, but in fact it carries the very important notion that modeling errors are at the same level as coding errors.

As a result of erroneous model, no code must be generated. It is important to always validate the model prior to generation – in case the model has been modified outside of the graphical editor. Error markers must be created in this case as well.

Content Assist and Automatic Completion

Hardly any developer will ever make do without content assist and automatic code completion anymore. Especially when modeling class-like structures, like we do in our business object domain, it is important to provide a similar feature in the graphical editor as well. The results are stunning. As expected, developers embrace this well-known feature and have no problems using it.

Automatic Code Generation

A major annoyance in MDSD is the explicit manual task of running the code generator. This is a tangible separation between modeling and coding. Moreover, it is not a part of the automatic (and incremental) compiler invocation that we're used to in the Java world.

We solved this problem by hooking the generator as an Eclipse builder into the regular automatic build process. The result is that whenever the model file (as specified in the generator configuration) is saved, the generator runs in a background task and creates the artifacts resulting from the model, validating the model on-the-fly. Source code that results from generation will be compiled immediately afterwards by the Java builder and can be used instantaneously. This provides an immediate round-trip from model to code.

Fast Model Navigation

While changes in the model are reflected in almost no time in the generated source code, we still do have a separation of model and code. As a general rule, generated source code must not be modified manually, since changes to the code are not reflected in the model. It is therefore imperative for the developer to make certain changes in the model. Often, the need for such changes arises while developing implementation code in Java classes. While navigation within Java code is nowadays made very easy by IDEs, finding model elements that need to be changed can be challenging, especially in large models.

A first, very obvious solution is to provide a model navigator view that allows navigating the model tree hierarchically, very similar to the package navigator view for Java. But once a developer knows the code he is working with, he hardly uses the navigator view anymore, but instead turns to the *Open Type…* dialog (defaults to the CTRL-SHIFT-T shortcut) which allows finding a known type by name, with wildcard and camel-case support.

We implemented a similar function with an *Open Model Element…* dialog (mapped to a CTRL-SHFT-Z shortcut). Instead of Java types, it displays the names of model elements, along with the diagrams in which they are visualized. If an element is shown in multiple diagrams, separate entries are shown in the dialog, similar to classes with the same name in separate packages. The model elements are read and cached in background jobs, so the lookup is kept fast.

Conclusion and Outlook

In this paper, we have shown that it is possible to integrate MDSD tools in order to provide a productive environment for the MDSD developer. The boundaries between modeling and coding are blurred by automatic code generation and fast navigation from code to model. Many problems that made developers reluctant to adopt MDSD have been solved.

Still, many things remain to be done. The generator itself, for example, currently performs a complete generation process for every model change. Although running in the background, this can be a performance problem for large models. Incremental generation, similar to incremental compilation is a topic of current research.

In terms of model navigation, an even more direct way from code to model, similar to the “F3” shortcut or “CTRL-Click” hyperlinking would be desirable. Implementing this feature, though, would require full traceability from model element via generator template to the resulting artifact (and back). This traceability is a research area as well.

Finally, one major productivity booster of recent years is automated refactoring. While it is, for instance, very easy to rename a model element and keep the model itself consistent, Java code that references artifacts generated out of that model element will still refer to the old element. Here, the separation of model and code is still clearly visible to the developer. Special template-aware refactoring of model elements is necessary to solve this problem.

Still, with all the steps yet to be taken, ubiquitous MDSD is within reach using the technologies we have at hand today.