

Adding Pluggable Meta Models to *FUJABA*

Tobias Röttschke
Fachgebiet Echtzeitsysteme
Institut für Datentechnik (FB18)
Technische Universität Darmstadt
Darmstadt, Germany

tobias.roetschke@es.tu-darmstadt.de

ABSTRACT

In this paper we propose to split the structural part of the Fujaba meta model in an internal, tool-specific and an external, standardized metamodel. External metamodels and adequate editors (and code generators) should be provided by plugins. By this means, Fujaba would gain flexibility with respect to evolving meta model standards and allow the Fujaba community to reuse the Fujaba graph rewriting engine for different schema definition languages. In our opinion, this proposal is a major contribution to strengthen the Fujaba community and keep the various development streams together.

1. INTRODUCTION

One of the most prominent features of Fujaba[21] is the ability to visually specify behaviour by means of graph transformations written in the *Story Driven Modelling (SDM)*[6] language. Unlike its predecessor PROGRES [19], Fujaba aimed for adopting standard modelling languages which are well-known to a large community of software engineers. Consequently, Fujaba uses a fixed UML-like meta model, which is referred to throughout the system. Graph schemata are defined by UML 1.x class diagrams, transformation are specified with slightly modified UML activity diagrams, collaboration diagrams, and state charts.

However, as new languages like MOF 2.0 [14] and UML 2.0 [16] come up, Fujaba should be able to evolve with these languages to stay competitive and hence avoiding one of the major drawbacks of PROGRES. The basic idea would be to separate the internal SDM meta model required for the graph rewriting engine from an external meta model to allow the user to specify in the modelling language of his choice. While the internal model belongs to the FUJABA core, the external meta model would be added using the FUJABA plugin mechanism.

During our ongoing effort to add a MOF 2.0 schema edi-

tor with JMI[5]-compliant code generation while still using SDM for behavioural specifications [1], we found it necessary to perform this separation and successfully managed to implement some critical parts of it. Our current FUJABA implementation allows the user to choose between the original UML 1.x meta model and a new MOF 2.0 meta model, edit class diagrams and generate appropriate code.

However, adopting our version would require plugin developers to adapt their plugins. As we try to stay as close to the original implementation as possible, the effort will be basically reduced to renaming meta model related type references. In our opinion the benefits of the proposed modification will clearly outweigh this inconvenience. So we hope that the meta model separation will be re-integrated in the main branch after the Fujaba Days.

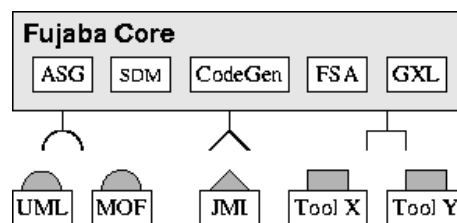


Figure 1: Our Fujaba vision

Figure 1 sketches our Fujaba vision. The Fujaba core contains all generic packages like *ASG*, *FSA*, *CodeGen*, and finally the internal *SDM* meta model. External meta models like *UML 2.0* or *MOF 2.0*, standardized code generators like *JMI* and – as before – various tools based on the Fujaba platform can be plugged into the system.

Section 2 describes the shortcomings of the current meta model. In section 3, we propose how FUJABA should be reengineered to be able to support pluggable meta models and code generators.

2. ABOUT THE FUJABA META MODEL

As discussed in section 1, Fujaba has a fixed meta model and code generator, which cannot be replaced easily. Besides, Fujaba lacks support for a standard application interface for (meta) modelling tools, for instance *JMI*. This has also been criticized in [18], where accordingly a new refactoring tool called *JMI Conform Model Transformator Generator (JCMTG)* is derived from Fujaba using a *MOF*

1.4 meta model [13] and the AndroMDA [3] pluggable code generator. In [1], another project is described, where Fujaba is adapted to a MOF 2.0 meta model and the MomoC code generator [2] is used. However, creating and maintaining a new tool for every meta model would not be wise, since there is a significant overlap between meta models and little variety with respect to SDM requirements. Instead it would be better to allow pluggable external meta models and code generators which are plugged into the Fujaba core using its own internal meta model.

In an ideal world, these features would have been part of the plugin mechanism introduced with version 4.0. Fujaba's key competence lies graph transformation engine, which distinguishes it from most other tool platforms. The current plugin mechanism allows only to build graphical editors based on the Fujaba meta model and code generator. But the true added value lies in application of the graph transformation engine to arbitrary meta models and code generators.

Apart from the above, presentation-related concepts of the Fujaba meta model like "project", "diagram" and "file" cannot be found in standard modelling languages like UML 1.x, UML 2.0 or MOF 2.0. In the UML 2.0 Diagram Interchange Specification [15] however, a meta model extension to UML is proposed, which allows to model presentation information. Unfortunately, it does not yet provide full support for UML 2.0 or MOF 2.0 and the document appears to be rather unfinished, although marked as "Final Adopted Specification". With respect to Fujaba, there should be a clear separation between standardized meta model elements and not yet standardized presentation elements. This would make it easier to adopt a designated standard early.

Looking into Fujaba in more depth, one finds that some classes are real god classes [17]. *UMLProject* for instance has almost 3000 lines of code and performs many tasks: Static methods for accessing administrative classes and performing string operations, complex algorithms for loading projects and finally, it acts as part of the Fujaba meta model. Accordingly, *UMLProject* is referred to throughout the whole system. Changing the meta model would hence result in countless modifications of the code.

Another drawback of UML Project is, that it only allows for one project instance. Especially when using Fujaba for model integration [7] or tool integration [4] purposes, it would make sense to work with multiple projects of different kinds. There would be one project for each integrated model or tool, which would manage its specific meta model, implements a specific algorithm to load or store model and so on. The integration part with its integration model should be an additional project depending on the model / tool specific projects.

Besides, the Fujaba meta model has some extensions for project definition and Java language artifacts, especially modifiers. Static classes [10], package visibility, final, native or synchronized methods are just a few examples. Apart from programming language specific visibilities [12], these artifacts are neither found in one of the official UML-meta models nor in the MOF meta models.

The conclusion is that the existing Fujaba meta model merges modelling language, programming language and project concepts into a single meta model. Assuming for the time being, that programming language and project concepts are fixed within the Fujaba context, it should still be possible to use different meta models for the modelling language, because evaluating and improving evolving modelling languages is one of the major research challenges that should be addressed by the Fujaba community.

Many features like behavioural diagram editors and the code generator are related to the existing all-in-one meta model. Our idea is to separate between an internal so-called SDM meta model and an external meta model, which could be UML 1.x, UML 2.0, MOF 2.0 or similar. As we want to *replace* parts of the meta model rather than *extend* it, the Fujaba plugin mechanism alone is not sufficient. On the one hand we would have to duplicate much of the Fujaba code to reuse the existing Fujaba platform when adding an extra meta model, and just writing another editor plugin for it. This would hamper Fujaba's maintainability. Extending the Fujaba meta model by means of inheritance on the other hand, would also not solve the problem, as most meta models obviously are not a superset of UML 1.4. Apart from the threat of name collisions, this would result in overly large classes and needlessly increase the runtime-overhead.

3. PROPOSED SOLUTION

This section describes, how we propose to reengineer the Fujaba core, so that different meta model plugins could be used with the Fujaba platform. First we describe, how *UMLProject* has to be refactored to allow for multiple, meta model-specific projects in a Fujaba session. Next, we present the designated internal Fujaba meta model before we explain, how an external meta model is linked to the internal meta model. Finally, we describe how plugin developers have to adjust their code to make the plugins work with the new Fujaba core.

3.1 Breaking up *UMLProject*

In our implementation, *UMLProject* is replaced by four new classes: *ProjectManager*, *ProjectLoader*, *SDMMetaModel*, and *SDMProject*. Figure 2 shows the interaction of these classes.

The *ProjectManager* takes over the static part of the former *UMLProject*. It is responsible for managing instances of projects, meta models, and loaders. The *ProjectLoader* is responsible for loading and storing Fujaba models. Each external meta model will usually have its own loader, but even different loaders could be combined with a meta model (e.g. XMI, GXL, FPR, or native tool specific loaders) following the *Strategy* [8] pattern. *SDMMetaModel* is responsible for creation of and access to model element instances. Most methods are abstract, as the concrete meta model plugin decides, how instances are managed. Concrete methods usually implement query algorithms or map model elements to Java entities, which is considered being part of the internal meta model. Finally *SDMProject* contains the purely project related stuff concerning diagrams, import/export and filesystem entities, which is not covered by an external meta model.

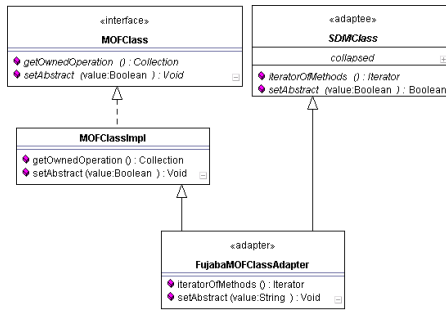


Figure 4: Applying the class adapter pattern

Both *MOFClass* and *SDMClass* provide an attribute *abstract*, which is accessed by identically named access methods called *setAbstract*. In *MOFClassImpl*, this method is implemented without being aware of the Fujaba platform. In *FujabaMOFClassAdapter*, this implementation is overridden by the following code:

```

void setAbstract (boolean value) {
    boolean changed = false;
    if (super.isAbstract() != value) {
        super.setAbstract(value);
        changed = true;
        firePropertyChange ("abstract", !value, value);
    }
    // return changed;
}

```

This example demonstrates, how generated and handwritten code can be combined nicely. But there is also a naming conflict as the return types of the *setAbstract* method in *SDMClass* and *MOFClass* do not match, while the signature is identical. We are currently working on systematic prefix conventions for the internal meta model to allow external meta models to be as close to the standard as possible. Covariant returns[9] as proposed for the next Java version, would provide a clean solution to this problem.

The following code fragment shows, how external and internal meta model can be connected, even if there are minor conceptual differences. However, attention must be paid to the correct usage of Fujaba *properties* as reflection occurs at many occasions. So when implementing the unparse module for *MOFClass*, the *ClassCompartmentVisibilityUpdater* for MOF operations must be initialized with the property “methods”, because the invocation of *iteratorOfMethods* is triggered by this name via reflection.

```

Iterator iteratorOfMethods {
    return super.getOwnedOperation().iterator();
}

```

Apart from the above, the internal and the external meta model do not always fit perfectly together. Some thought has to be put on the mapping of *MOFAssociation* to *SDMAssoc* as MOF associations are far more powerful than former

UML associations. But as most of the MOF-specific functionality is hidden in the code generated by the MomoC tool, adaption should be possible. Somewhat more inconvenient is the fact that an instance of *MOFProperty* corresponds to instances of *SDMAttr*, *SDMRole* and *SDMCardinality*. In this case, we use the Object Adapter pattern [8] rather than the Class Adapter pattern, which results in slightly more runtime objects. *MOFElementImport*, which is used to create references to elements from foreign *MOFPackage* instances, acts as *SDMClass* or *SDMAssoc* from the Fujaba point of view and can be realized as object adapter as well.

3.4 Impact on other plugins

The increased flexibility of our proposal comes at the cost of some refactoring effort for plugin developers. Fortunately, the modifications are rather trivial.

As the Fujaba core does not contain a concrete meta model implementation anymore, most plugins will depend on a meta model plugin like the UML- or MOF-plugin. As *UML-Project* has been redesigned, calls of *UMLProject.get()* have to be replaced by *ProjectManager.getProject(...)*. Although not yet implemented, it would be easily possible to allow the *ProjectManager* to handle multiple projects, each depending on a different meta model plugin. References to the internal Fujaba meta model using *UML...* have to be renamed into *SDM...* The prefix “UML” would refer to an external meta model imported by the UML-Plugin. When creating a new instance of a model element inside the core, it is no longer allowed to directly call the constructor of this element. Instead of calling for instance

```
clazz = new UMLClass(name);
```

one would have to call

```

SDMProject project = ProjectManager.getProject();
SDMMetaModel model = project.getMetaModel();
clazz = model.createClass();
clazz.setName (name);

```

This might look more complicated initially, but *SDMProject* and *SDMMetaModel* are referred to very often throughout the code and hence readily available. In practice, only a few extra lines have to be added. For the sake of backward compatibility, the UML meta model plugin will still provide the old constructors, so that existing tool plugins are easily adapted.

4. CONCLUSION

In this paper we have proposed, how to split the Fujaba meta model into an internal SDM meta model and standard-compliant external meta models that can be provided by plugins. We actually consider this a logical and necessary step, which should already have been implemented by the plugin mechanism introduced quite recently. We have pointed out that the Fujaba community will benefit from keeping up with evolving meta model specifications, without the necessity to modify the core. Considering our current needs,

external meta models have been restricted to structural diagrams, while behavioural diagrams are reused based on the internal meta model.

We are currently implementing the proposed changes in the “Refactoring” branch of the Fujaba CVS repository and prepare a demo of the modified Fujaba core together with a UML and a MOF plugin. Our effort is driven by the desire to use Fujaba as MOF 2.0 editor and JMI code generator. Based on the original Fujaba core, our goal could not be reached, as the Fujaba code generator which has been used to bootstrap Fujaba violates JMI guidelines beyond repair. Similar problems have already forced the JCMTG project to separate from the Fujaba development stream. We are convinced that our proposed changes are necessary to keep the Fujaba community together and improve the maintainability of the code base.

From our experiences with transforming the Fujaba core and adapting the MOF plugin, we estimate the migration effort for plugin developers to be approximately one or two days. Some additional effort would be required to integrate all modifications to the main branch since the refactoring branch has been created. The sooner both branches are merged the better. Together with the UML-Plugin, the refactoring branch should provide the same features as the main branch.

Currently, there are only two major concerns: On the one hand, Java lacks multiple inheritance between classes, resulting in either more runtime objects or code duplication. On the other, Fujaba and MOF/JMI use different approaches for association implementation, which are not easily coupled.

The ideas presented here would allow to use the Fujaba code generator for different meta models. Consequently, we will investigate how pluggable code generators (e.g. JMI compliant) can be realized based on the proposed modifications. As the existing code generation concepts are already rather flexible, we are confident that pluggable code generators can be realized with reasonable effort.

5. REFERENCES

- [1] C. Amelunxen. MOF 2.0 Editor Plugin for Fujaba. In Schürr and Zündorf [20]. To appear.
- [2] L. Bichler. Tool Support for Generating Implementations of MOF-based Modeling Languages. In J. Gray, J.-P. Tolvanen, and M. Rossi, editors, *Proceedings of the 3rd OOPSLA Workshop on Domain-Specific Modeling*, Anaheim, California, USA, October 2003.
- [3] M. Bohlen. *AndroMDA: From UML to deployable components*, 2004. <http://www.andromda.org>.
- [4] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In *Proc. Workshop on Tool Integration in System Development*, Helsinki, Finland, September 2003.
- [5] R. Dirckze. *Java Metadata Interface (JMI) Specification, v1.0*. Unisys Corporation, Sun Microsystems, Inc., June 2002. <http://java.sun.com/products/jmi/>.
- [6] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Grammar Language based on the Unified Modelling Language and Java. In *Workshop on Theory and Application of Graph Transformation (TAGT'98)*. University-GH Paderborn, Nov. 1998.
- [7] R. Freude and A. Königs. Tool Integration with Consistency Relations and their Visualization. In *Proc. Workshop on Tool Integration in System Development*, Helsinki, Finland, September 2003.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd edition draft*, 2004.
- [10] JavaWorld. Static class declarations, 1999. <http://www.javaworld.com/javaqa/1999-08/01-qa-static2.html>.
- [11] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes on Computer Science*. Springer, 1996.
- [12] Object Management Group, Inc. *OMG Unified Modeling Language Specification Version 1.4*, Sept. 2001. <http://www.omg.org/docs/formal/01-09-67.pdf>.
- [13] Object Management Group, Inc. *Meta-Object Facility (MOF) Specification Version 1.4*, Apr. 2002. <http://www.omg.org/docs/formal/02-04-03.pdf>.
- [14] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, Oct. 2003. <http://www.omg.org/docs/ptc/03-10-04.pdf>.
- [15] Object Management Group, Inc. *UML 2.0 Diagram Interchange Specification*, Sept. 2003. <http://www.omg.org/docs/ptc/03-09-01.pdf>.
- [16] Object Management Group, Inc. *Unified Modeling Language (UML) Specification: Infrastructure Version 2.0*, Sept. 2003. <http://www.omg.org/docs/ptc/03-09-15.pdf>.
- [17] A. J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.
- [18] H. Schippers. JMI Conforme Modeltransformator-Generator. Master’s thesis, Universiteit Antwerpen, 2004. In Flemish.
- [19] A. Schürr, A. J. Winter, and A. Zündorf. Developing Tools with the PROGRES Environment. In Nagl [11], pages 356–369.
- [20] A. Schürr and A. Zündorf, editors. *Fujaba Days 2004*. TU Darmstadt, 2004. Technical Report. To appear.
- [21] A. Zündorf. *Rigorous Object Oriented Software Development*. Universität Paderborn, 2001. Habilitation Thesis.