

A language to describe software texture in abstract design models and implementation

Joern Bettin
Equinox Ltd. Software Architects
Level 12, Equinox House, 111 The Terrace,
PO Box 10 168
Wellington, New Zealand
+64 4 499 9450
joern.bettin@equinox.co.nz

Abstract

A model-driven software development approach has been used in a project to build a complex commercial application within the New Zealand electricity industry. As part of the project a compact visual notation for the mapping between a highly abstract UML design model and implementation has been developed. The overall approach makes use of modelling and meta-modelling techniques, formal specifications of component architecture standards, a commercial UML modelling tool, and a template-based Java code generator.

1. UML design models

Most commercial UML modelling tools promote modelling at the same level of abstraction as the implementation. Essentially this approach interprets the UML as a graphical notation that provides a view into the implementation source code. In particular if the round-trip-engineering features of the typical commercial UML tools are used, the resulting models contain repeated instances of design patterns that have been used in the implementation. This use of the UML can lead to frustrated development teams where the UML tool is mainly used as a drawing tool for post-implementation documentation.

Our motivation to use the UML is congruent with [Harrison-00], where the UML design models are implementation language independent, and where the model can be used to generate implementation structures in potentially more than one target language. The achievable difference in the level of abstraction between UML model and implementation code directly corresponds to a gain in productivity, and depends to a large degree on the quality of the code generation techniques used to map from UML model to target language(s).

The abstract UML models can be used as reliable documentation by subsequent projects. Through the use of appropriate code generation techniques the models can be re-used in other projects in significantly different target implementation environments.

2. From design to implementation

Techniques

For maximum clarity and precision we chose a compact representation of the mapping between textures in the UML design model and textures in the implementation. The basic idea is straight forward: use the UML to model each *allowable* construct that can appear in the UML *design model* in an *architecture model*, and then use the UML to model all the corresponding Java implementation constructs in an *implementation model*.

In the design model (figure 1) we use stereotypes in the usual way to indicate the role of the individual UML classes.

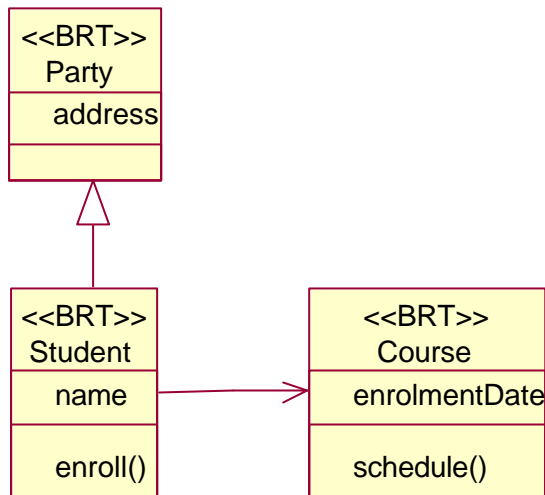


figure 1.

The UML has several shortcomings when used as an architecture description language. We resorted to what could be called *multi-level stereotyping*. In the architecture model (figure 2) we use a `<<Stereotype>>` stereotype to label a class `Foo` which represents all classes of stereotype `<<Foo>>` in the design model. Similarly, in the architecture model we use the `<<Stereotype>>` stereotype to label an operation `bar` of class `Foo` which represents all operations of stereotype `<<bar>>` of all classes of stereotype `<<Foo>>` in the design model.

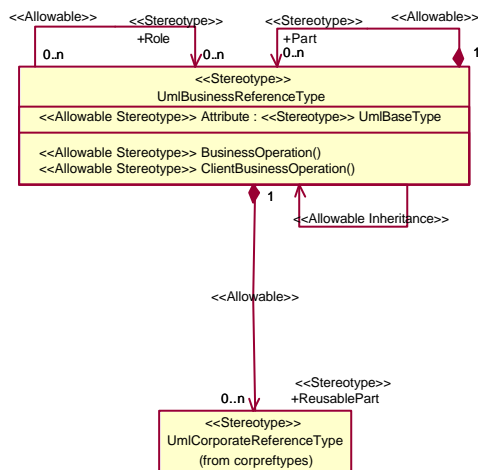


figure 2.

In the UML each model element is only allowed to be assigned to one stereotype. We however need *multi-dimensional stereotypes* to express the mapping between design model and implementation model with sufficient precision. Tagged values could be used to classify model elements along multiple dimensions, but we already use tagged values to capture additional model element properties required to fine-tune code generation, and prefer the visual UML notation of stereotypes. We use the `<<Allowable>>` stereotype with operations, links, and attributes to denote items in the architecture and implementation models that are *allowable constructs*. Conversely all items not labelled with the `<<Allowable>>` stereotype represent *mandatory constructs*.

For example class `<<Stereotype>> Foo` in the architecture model may inherit from class `SuperFoo`, meaning that in the design model every class `<<Foo>> DesignFoo` must inherit from `SuperFoo`. If class `<<Stereotype>> Foo` in the architecture model has an `<<Allowable>>` association with class `<<Stereotype>> Bar` this means that in the design model a class `<<Foo>> DesignFoo` may have an association with `<<Bar>> DesignBar`.

The semantics of the `<<Stereotype>>` and the `<<Allowable>>` stereotypes are in conflict with the “validation rules” enforced by most UML tools, such as checks for cyclical inheritance relationships. In this and in similar situations we have used stereotyped dependency links to model the construct that our UML diagramming tool did not allow.

The same modelling techniques are used in the implementation model (figure 3), to the effect that for each UML diagram in the architecture model there is a corresponding UML diagram in the implementation model. The mapping between both models is established by a simple set of strict mapping rules for model element names. For example all implementation classes and interfaces corresponding to a class `Foo` in the architecture model are named `Foo<suffix>` in the implementation model.

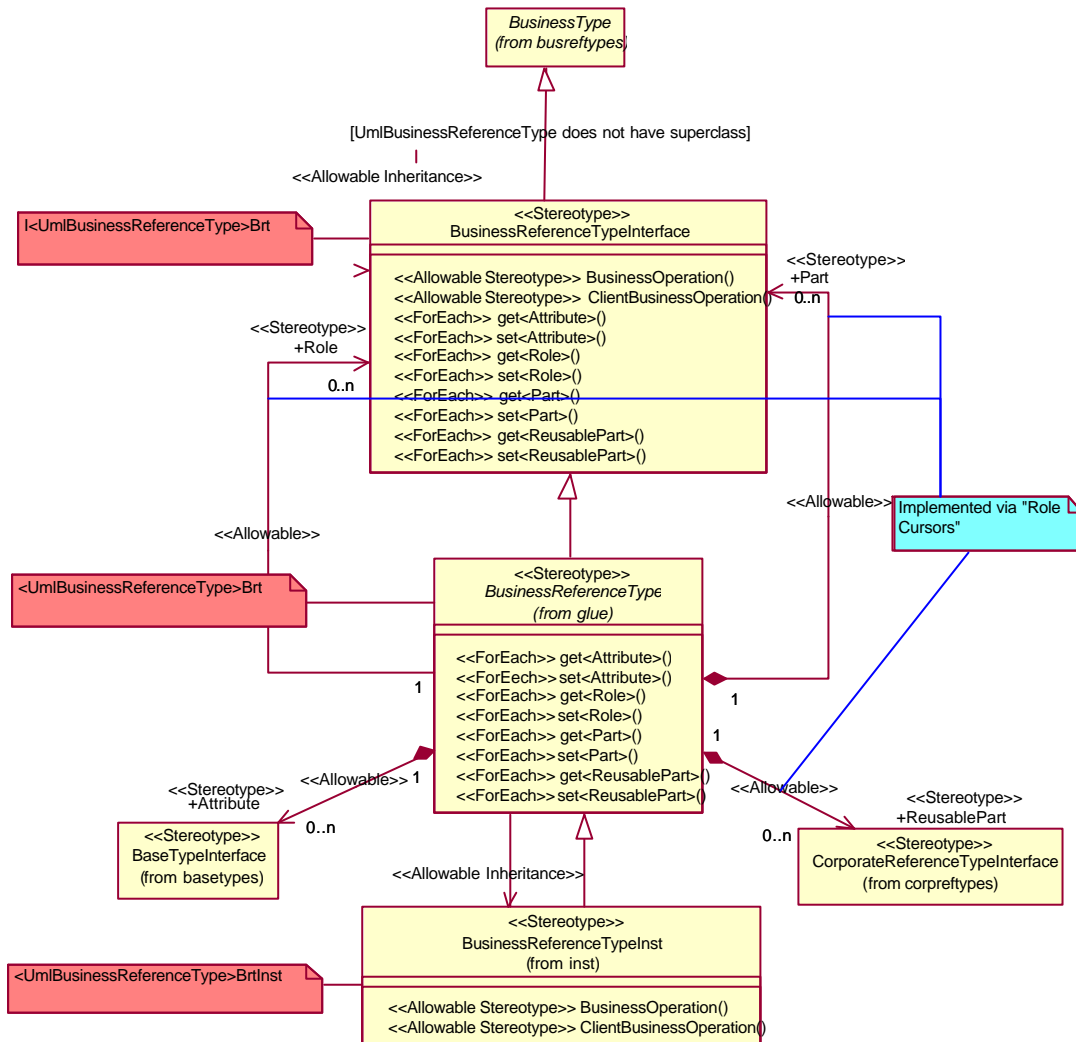


figure 3.

In summary our modelling and meta-modelling techniques lead to a highly compact and precise specification of implementation patterns. The abstract nature of the notation however raises issues with using UML tools that only have restricted meta-modelling capabilities. In other words the UML extension mechanisms of stereotypes and tagged values are insufficient to allow usage of the UML as a complete architecture description language.

Mapping to a Java, XML and SQL implementation

The actual mapping of the architecture model to the implementation model makes use of patterns similar to those described by [Harrison-00] supplemented by additional patterns that reflect the project-specific application architecture. Most classes in the architecture model correspond to a pattern of a Java interface, an abstract Java class, a concrete Java class, and a Java collection class [used for the implementation of associations], SQL DDL [for persistent classes], and corresponding XML definitions in the implementation.

3. Improving the notation

There are several weaknesses with the notation described in the previous section. We are looking for a visual notation that captures the *mapping* of software textures in a design model to software textures in the implementation. The UML only offers *traces* relationships between model elements of type “class”, we however require a similar construct that can be used to map arbitrary model elements including operations, attributes and links. To do so we invent a *texture diagram* notation that allows us to map architectural textures to implementation textures in one diagram. The underlying meta-model should allow the implementation of automated architecture compliance checks.

We also choose to remove our use of stereotypes and replace it with specific visual icons that can be attached to model elements. Finally we realise that the texture diagrams can also capture the mapping of corresponding model element names in the design model and the implementation in a formal notation. This eliminates the use of UML notes to capture “naming rules”.

Texture diagrams

The notational elements required are the following:

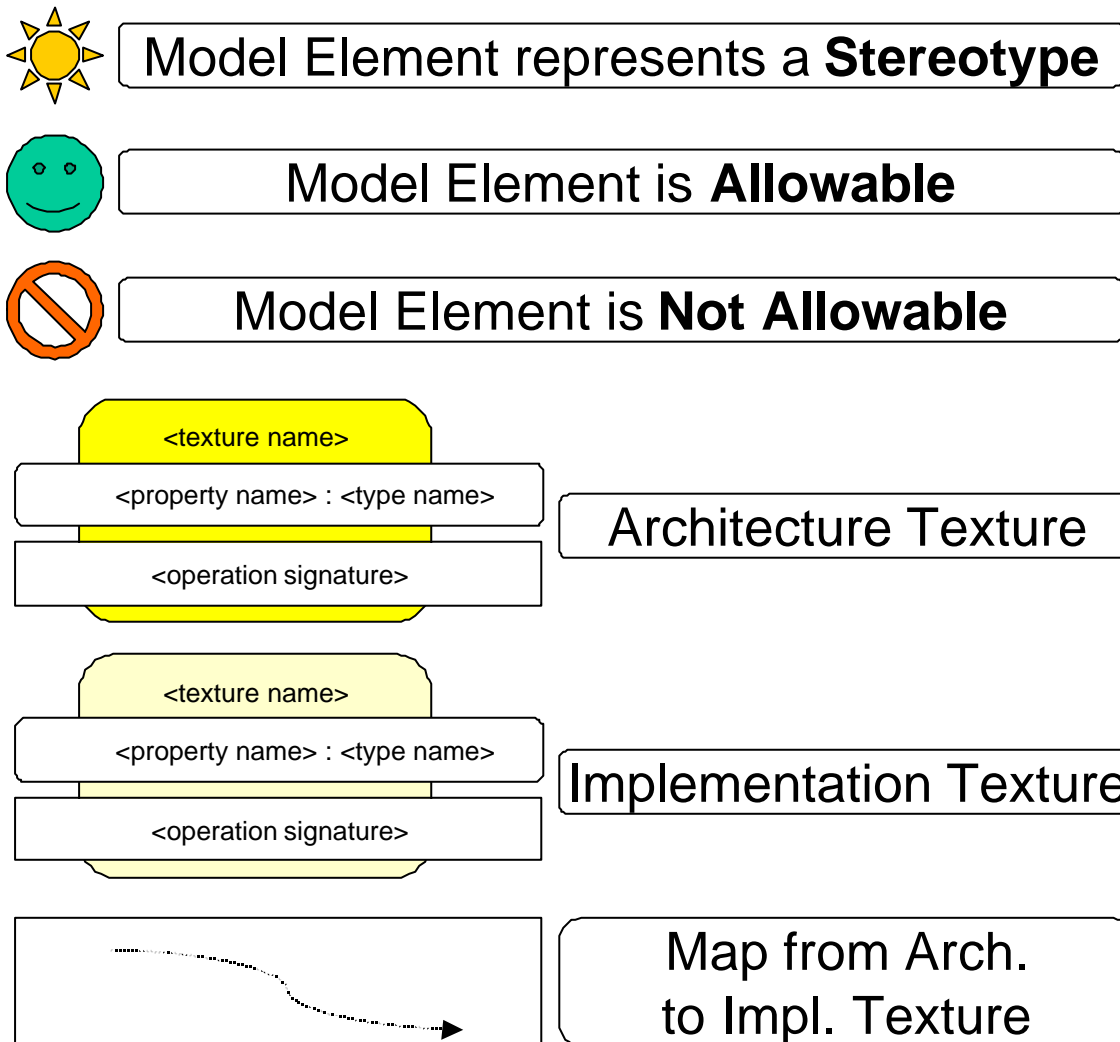


figure 4.

Note that in our particular application of texture diagrams, textures happen to be classes – this does not have to be the case. The mapping that was implicitly expressed in figures 2. and 3. is now explicitly expressed in figures 5. to 7. The visual advantages of the new notation are apparent. Behind the scenes the other advantage is a clean meta model that stores the texture-mappings in a machine-readable format.

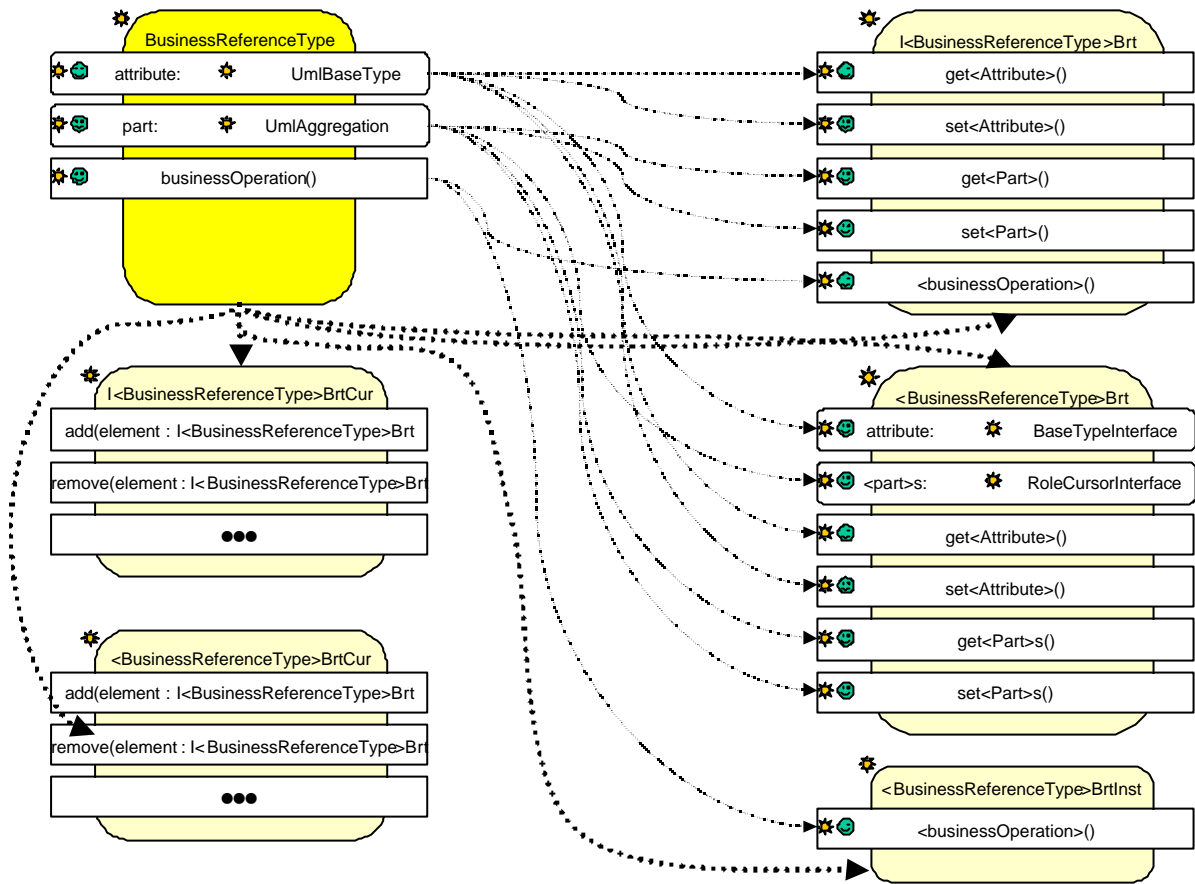


figure 5. "Class Texture"

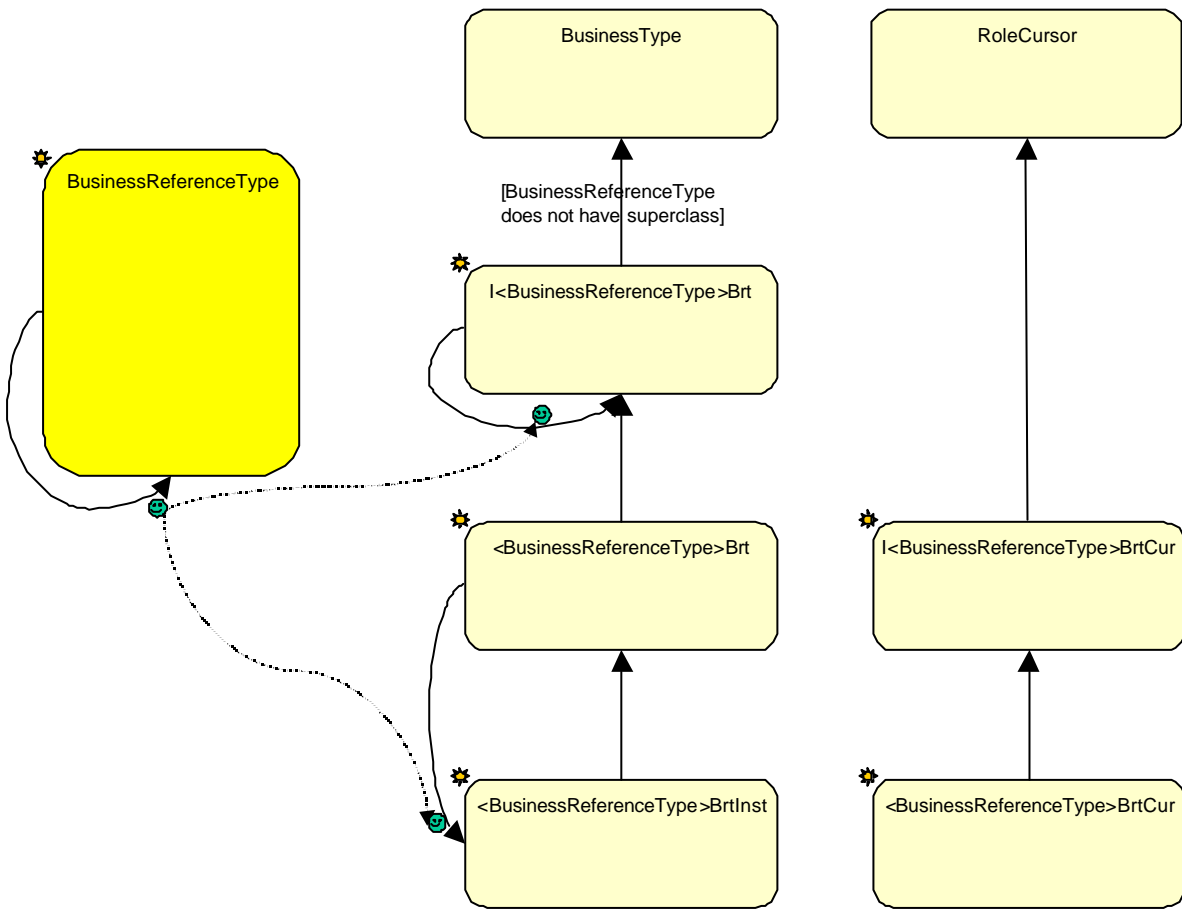


figure 6. "Inheritance Texture"

