PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 44, NO. 2, PP. 141–157 (2000)

CODE GENERATION FROM UML MODELS[1]

Ákos FROHNER, Zoltán PORKOLÁB and László VARGA

Department of General Computer Science
Eötvös Loránd University, Budapest, Hungary
e-mail: Akos.Frohner@elte.hu, Zoltan.Porkolab@elte.hu, varga@ludens.elte.hu

Received: Nov. 3, 2000

Abstract

Creating a generic, object-oriented, component-based, transactional business system, which covers the whole lifecycle, is possible only with the integration of commercial tools, component technologies, newly developed class libraries and by using code generators. Most of the recently used tools for development techniques are focusing on only one of the layers of the model from the code generation point of view. As a consequence, the inter-layer connections are lost in the generated code.

In this article, we describe a code generator technique which uses a UML model as a starting point and generates several layers directly. While generating the code, it preserves the original inter-layer relationships originated in the model.

Based on our experiences with 4GL systems it is obvious that there is a need to provide customisation in the generated code. We offer a multi-paradigm approach [1] to let the developer choose the appropriate solution for her or his implementation.

Keywords: UML, code generation, component, metadata, aspect-oriented programming.

1. Introduction

Nowadays, new programming paradigms have come into view in software technology. Multi-Paradigm Design [2] by James O. COPLIEN and others, focuses on helping the designer to create abstraction for arbitrary domains. Object-oriented design arms the designer with tools that produce modules of a certain shape. As long as the problem domain lends itself well to object-shaped abstractions, the object paradigm works well. However, some problems have little to do with objects. Multi-paradigm steps above any single paradigm as it helps the designer choose the right paradigm for each project domain.

We did not intend to limit the expressiveness of a designer with only one paradigm, thus we show various solutions with various paradigms where it is applicable.

Shortly, we introduce the following paradigms to help the understanding of the *customisation techniques* (see Section 4):

- Generative Programming
- Generic Programming
- Aspect Oriented Programming

[1]This work has been supported by the Grants OMFB ALK-00229/98

Object-oriented programming is also heavily used, but it is a widely known paradigm, so we expect that the reader's knowledge is sufficient to follow the object-oriented customisation techniques used in this article.

1.1. Generative Programming

Generative Programming (GP) is a software engineering paradigm based on modelling software system families such that, given a particular requirements specification, a highly customised and optimised intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [3].

In this article we use the idea of GP in various places. At the design level, we generate sub-layers of the analysis model. At implementation, we generate client code from UI design models, and database schema from the model of the persistent layer.

1.2. Generic Programming

Of different programming paradigms the most interesting new paradigm is the generic programming. The goal of this style is to reveal the foundations and programming methods of generic, and therefore reusable, components and libraries. In the terms of Multi-Paradigm theory we use generic programming in those cases when the objects have little or no common structure but the behaviour (the methods used on objects) are similar. Using this approach we can greatly reduce the complexity of a software library. For example, if we have a library with n data structures, each with k base type and m algorithms, then using the traditional object-oriented way the complexity of the library is $O(k * n * m)$. The using of the generic programming this reduced to $O(n + m)$ [5].

The software library designs that have resulted from this generic programming approach are markedly different from other software libraries: the precisely-organized, interchangeable building blocks that result from the approach permit many more useful combinations than are possible with more traditional component designs. The design is also a suitable basis for further development of components for specialized areas such as databases, user interfaces, and so on. By employing compile-time mechanisms and paying due regard to algorithm issues, component generality can be achieved without sacrificing efficiency. This is in sharp contrast to the inefficiencies often introduced by other library structures involving complex inheritance hierarchies and extensive use of virtual functions [4]. The bottom-line results of these differences is that generic components are far more useful to programmers, and therefore far more likely to be used, in preference to programming every algorithm or data structure operation from scratch.

1.3. Aspect-Oriented Programming

4 Traditionally, programs involving shared resources, multi-object protocols, error handling, complex performance optimisations and other systemic, or cross-cutting concerns have tended to have poor modularity. The implementation of these concerns typically ends up being tangled through the code, resulting in systems that are difficult to develop, understand and maintain.

Aspect-oriented programming is a technique that has been proposed specifically to address this problem. One can separate the above mentioned concerns on source level into aspects and weave them into the original code using an automated tool before compilation. The granularity of the weaving points and the language of the aspect is determined by the actual implementation of the weaver.

For implementation purposes we have chosen Java, since the parallel constructs are a standard part of the language and there is a general purpose aspect-weaver called AspectJ. This tool has been developed in the last couple of years at Xerox Palo Alto [14].

In AspectJ, aspects are programming constructs that work by cross-cutting the modularity of classes in carefully designed and principled ways. So, for example, a single aspect can affect the implementation of a number of methods in a number of classes.

1.4. Unified Modelling Language

The Unified Modelling Language [6] is a general-purpose visual modelling language that is designed to specify, visualise, construct and document the artifacts of a software system. The UML is simple and powerful. The language is based on a small number of core concepts that most object-oriented developers can easily learn and apply. The core concepts can be combined and extended so that expert object modellers can define large and complex systems across a wide range of domains.

In this paper we use only a small subset of the whole modelling language: static class diagrams and some semantical constructs. The used notation is in the full specification [13].

1.5. XMI

UML is sufficient to visualise the design, but it was necessary to develop a standard which enables the exchange of meta-information (**X**ML **M**etadata **I**nterchange) among design, implementation and runtime systems. OMG has introduced XMI [16] for this purpose.

Also, XMI is intended to be a ‘stream’ format. That is, it can either be stored in a traditional file system or streamed across the Internet from a database or repository.

In this article this format is used to export the design data from any OOAD design tool and base the implementation of the model transformation and code generation facilities on its model-architecture.

2. Design Model

The classical approach of software engineering identified the software life-cycle as a rigid sequential process: it starts with the analysis phase followed by the separate design and implementation steps. Nowadays, this bounded approach is no longer tenable. The accelerated process of software manufacturing requires iterative and sometimes even parallel execution of the analysis – design – implementation steps. However, this iterative/parallel software engineering process is not supported efficiently by recent CASE tools.

We intend to simplify and make this process partly automated by generating *layer specific* design models. Layer specific models are views within the original model specialized for the target domain of some software technology step, (i.e. analysis, database design, UI design etc.). To allow iterative and parallel modifications at each level, we give a method to *re-generate* the layer specific design models.

We give a formal description of the process with examples.

As an example application we have chosen a simple *car retail system* which was extended by various aspects at the analysis and design level.

2.1. Analysis

In the analysis level new classes are added to describe the usage patterns (e.g. 'in which garage is the car parked?', 'who is the test driver of a new model?') in the company. Further analysis steps also removed a class, which was unnecessary at this level (engine).

In the database design new classes are added to more precisely specify the car itself (e.g. the type of engine), while also considering some of the existing details of the persistent layer (e.g. records of cars with diesel engine are stored in a separate table than others). Another class is removed, which was not necessary from this point of view (person).

The first figure (see *Fig. 1*) shows the first version of the analysis model, with the following classes: *Vehicle, Car, Person, Engine*.

2.2. Initial Generation of the Layers

In the first phase the layer specific *sub-models* are generated within the original design model to represent the specific *layers*.

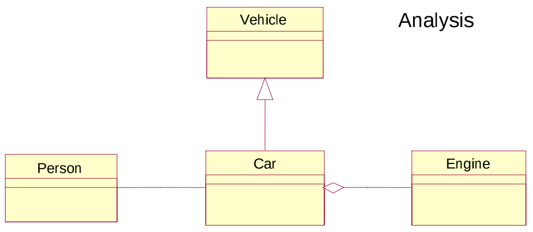

Fig. 1. Analysis - version 1

The sub-models are specialised for the target domain: they contain interfaces, classes and methods specific to the target language:

- Database or persistent layer
- Business logic
- Middle tier
- User interface (view, list, editor)

Fig. 2 illustrates the generation of such sub-models in our example car retail system.

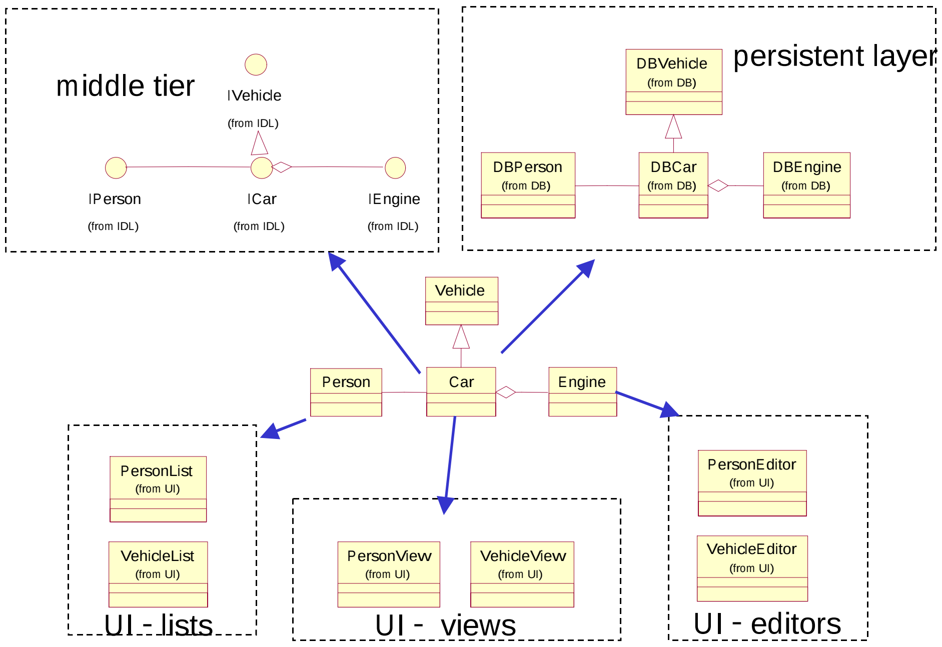

Fig. 2. Generated model layers

2.3. Re-generation Algorithm

The sub-model generation itself is a platform specific process which needs a lot of design-style specific decisions, thus we do not intend to describe the details.

The reusable part of the process is the algorithm which merges the modifications made at different layers. Once a layer specific sub-model is generated, the designer may change certain classes and relationships both in the original and in the generated layer. The re-generation algorithm shall transfer the changes from the original layer to the generated layer, meanwhile preserving as much of the modifications as possible.

To describe the algorithm formally we need to introduce several notations:

$$\begin{aligned} G^k &= \langle V^k, E^k \rangle, && (1)\\ V^k &= \{\textit{nodes of layer } k\}, && (2)\\ E^k &\subseteq V^k x V^k. && (3) \end{aligned}$$

For example, the first version of the analysis model is G_0^A, while the subsequent versions are marked with G_i^A. From the i^{th} version of the analysis model the generated layer k is marked by G_i^k.

$$\begin{aligned} Gen^k &: G^A \mapsto G^k, && (4)\\ DiffGen &: G^A x G^A x G^k \mapsto G^k, && (5)\\ G^k &= DiffGen(G_{i-1}^A, G_i^A, G_{i-1}^k), && (6)\\ DepNodes &: V \mapsto 2^V, && (7)\\ DepNodes(v) &= \{w \in V \mid w \textit{ depends-on } v\}, && (8)\\ DepEdges &: V \mapsto 2^E, && (9)\\ DepEdges(v) &= \{\langle v, w \rangle \in E \mid \forall w \in V \wedge v \in DepNodes(v)\}, && (10)\\ DepSet(\langle V, E \rangle) &= \{\langle DepNodes(v), DepEdges(v) \rangle \mid v \in V\}, && (11)\\ DelSet_i^k &= Gen^k(G_{i-1}^A \setminus G_i^A) \cup (Gen(G_{i-1}^A) \setminus G_{i-1}^k), && (12)\\ \overline{DelSet_i^k} &= DelSet_i^k \cup DepSet(DelSet_i^k), && (13)\\ DiffGen(G_{i-1}^A, G_i^A, G_{i-1}^k) &= (Gen(G_i^A) \cup G_{i-1}^k) \setminus \overline{DelSet_i^k}. && (14) \end{aligned}$$

The algorithm is basically a generic graph transformation process, so it can be applied on different levels of the model graph.

The UML static design model is categorised into packages, containing classifiers (class, interface or data type), which are made of features (attribute or method).

On the macroscopic level the *graph* (1) is made of **packages**, where the *edges* (3) are the *import* dependencies among them. The *depends-on* (8) relationship simply maps to the edges in the formal algorithm.

On the level of **classifiers** the nodes are *classes* and *interfaces*, where the edges are *inheritance*, *association* and *implements* relations. The *depends-on* relationship maps to *is-a* (inheritance), *has-a* (aggregation) and *implements* (class implements an interface) relations.

On the microscopic level of **features** the nodes are *attributes*, *methods* and *classifiers*, where the edges are containment and type relations. The *depends-on* relationship is quite complex:

- an attribute *depends-on* a classifier of its type
- a method *depends-on* a classifier type of any of its parameter or return types
- a feature (method or attribute) *depends-on* the classifier, where it is defined

In the actual *sub-model re-generation process* these levels of the algorithm are sequentially applied on the model, achieving full refinement step-by-step.

2.4. Re-generation Example

To clarify the formal algorithm the following example demonstrates one step of the process on one level, with the sub-model of the database layer. To have an overview of this step, with all the involved classes see *Fig.* 3.

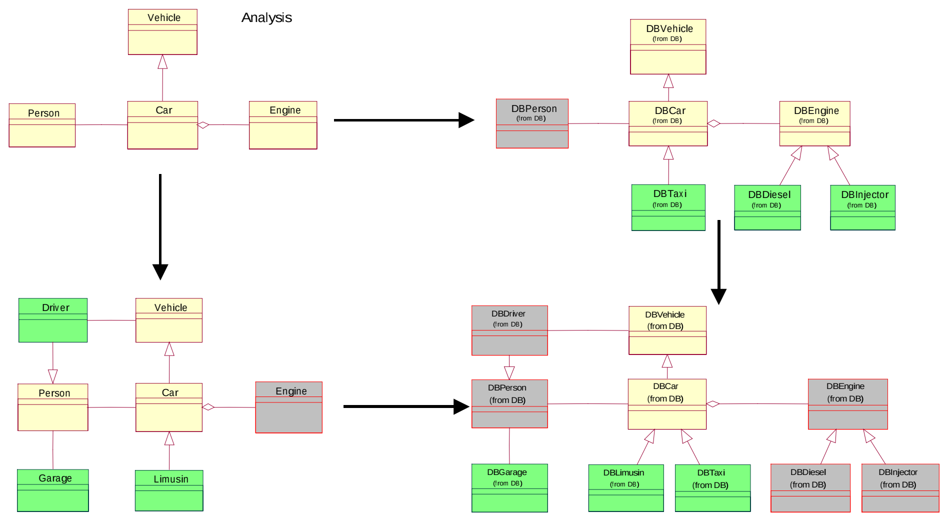

Fig. 3. Model generation - Overview

The starting point of the sub-model generation is the analysis model (see Section 2.1), the static class structure of a car retail system.

In the analysis, the following classes were identified: *Vehicle*, *Car*, *Engine* and *Person*, which can be formalised into the following set of nodes and edges:

$$V_0^A = \{Vehicle, Car, Engine, Person\},$$
$$E_0^A = \{\langle Person, Car\rangle, \langle Car, Engine\rangle, \langle Vehicle, Car\rangle\}.$$

2.4.1. Modifications in the Database Layer

The generation of the database layer's sub-model produces the following set of nodes:

$$V_0^{DB} = \{DBVehicle, DBCar, DBEngine, DBPerson\}.$$

The generated classes in the database layer may contain specific information for the target domain, such as

- omitted methods (no functionality is necessary in the database)
- changed type of attributes (int $\rightarrow$ NUMBER(38), string $\rightarrow$ VARCHAR(255))
- special fields (eg. object_id)

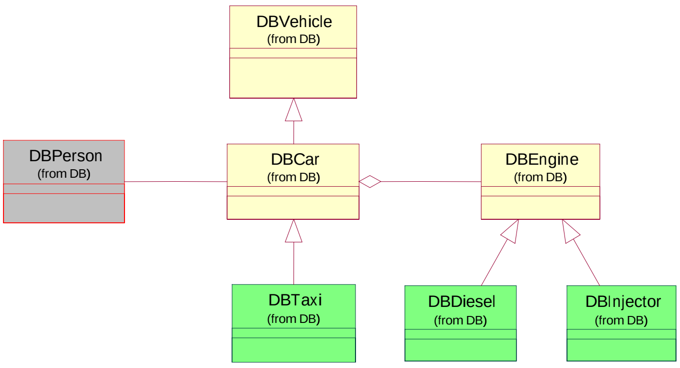

Fig. 4. Generated DB layer

During the refinement of the database scheme it was realised that the two types of engine (diesel and injector) cannot be fit into the same table, thus new classes are introduced as subclasses of *DBEngine*: *DBDiesel* and *DBInjector*. There were also special requirements for taxis (eg. an increased number of obligatory check in servicing for warranty), which implied the introduction of the *DBTaxi* class. The database designers were unconcerned of the person associated with the car, thus the *DBPerson* class was removed from this layer.

After these modifications the changed set of nodes and classes are the following:

$$\begin{aligned} V_0^{DB} &= \{\mathit{DBVehicle, DBCar, DBEngine, DBTaxi, DBDiesel, DBInjector}\},\\ E_0^{DB} &= \{\langle \mathit{DBCar, DBEngine}\rangle, \langle \mathit{DBVehicle, DBCar}\rangle,\\ &\quad \langle \mathit{DBCar, DBTaxi}\rangle, \langle \mathit{DBEngine, DBDiesel}\rangle,\\ &\quad \langle \mathit{DBEngine, DBInjector}\rangle\}. \end{aligned}$$

2.4.2. Modifications in the Analysis Model

Parallel to the development, further interviews with the customer may change the original analysis model.

In the case of the car retail system, the customer emphasised the need of dealing with situations where he has to know more about ‘where is a test car parked?’ and ‘who would drive a test car?’. The analysis model is changed by introducing the *Garage* and *Driver* classes.

To identify the type of a given car it was required to describe its general style instead of the details, thus the *Engine* class is dropped and the *Limusin* class is introduced.

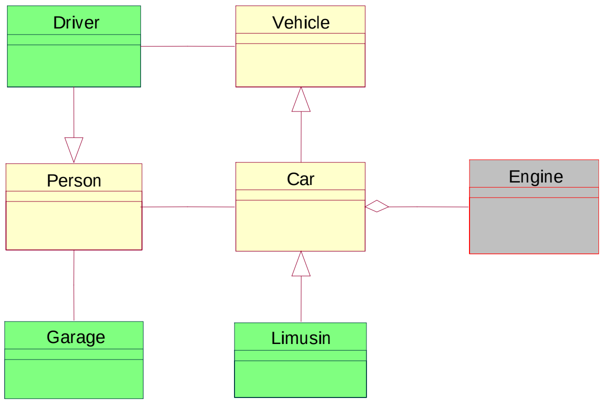

Fig. 5. Analysis - version 2

The changed set of nodes is:

$$V_1^A = \{\mathit{Vehicle, Car, Person, Driver, Garage, Limusin}\}.$$

2.4.3. Re-generation of the Database Layer

After the changes in the analysis and database layer, it is necessary to execute the re-generation process to have a consistent design model.

Some intermediate sets of the formal re-generation process (the edges are omitted for simplicity):

$$\mathit{DelSet}_1^{DB} = \langle\{\mathit{DBEngine}, \mathit{DBPerson}\}, \{\ldots\}\rangle, \tag{15}$$

$$\mathit{DepSet}(\mathit{DelSet}_1^{DB}) = \langle\{\mathit{DBDiesel}, \mathit{DBInjector}, \mathit{DBDriver}\}, \{\ldots\}\rangle, \tag{16}$$

$$\overline{\mathit{DelSet}_1^{DB}} = \langle\{\mathit{DBEngine}, \mathit{DBPerson}, \mathit{DBDiesel}, \mathit{DBInjector}, \mathit{DBDriver}\}, \{\ldots\}\rangle, \tag{17}$$

$$V_1^{DB} = \{\mathit{DBVehicle}, \mathit{DBCar}, \mathit{DBTaxi}, \mathit{DBLimusin}, \mathit{DBGarage}\}. \tag{18}$$

There are two deleted nodes (15): the *DBEngine* class was deleted in the database layer itself; the *Person* class was deleted in the analysis model, but the original generation process produced its counterpart as *DBPerson*.

Three classes are dependent upon the deleted ones (16): *DBDiesel* and *DBInjector is-a DBEngine*; *DBDriver is-a Person*.

The closure of the deleted node set is (17), so the new version of the database layer's sub-model consists of only five nodes (18).

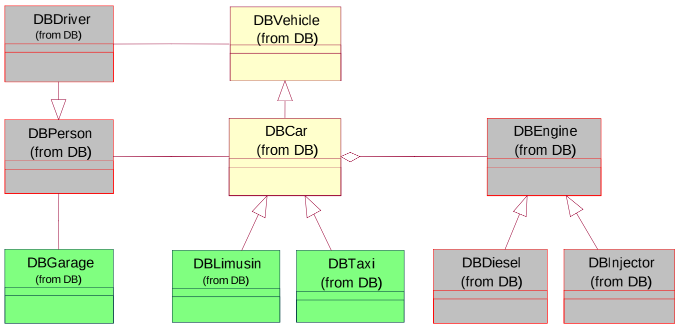

Fig. 6. Re-generated DB layer

3. Source Code Generation

The source codes shall be generated from the sub-models instead of the analysis model itself. The generated sources are platform specific, thus a sub-model can be the origin of more than one source code in different languages.

The usual targets are:

- SQL database schema

- Java or C++ for business logic
- Java or C++ for user interface
- meta information for various generic drivers
- IDL for the middle tier

The generated sources might be modified by the programmers to tailor them to the actual requirements. Unfortunately, these modifications are hard to track, thus it is hard to feed them back into the design model.

Two approaches can be used to deal with this problem:

1. *Reverse engineering*: parsing the modified source codes and building up the modified design model.
2. *Re-generation of the sources*: re-generation of the sources from the modified design model and merging the changes from both evolution paths in the same manner as the sub-models are generated and re-generated.

The sub-model generation algorithm can be implemented as a transformation on the model graph, but one does not have this clean architecture if the target graph is projected into source code. Although it is possible to reverse engineer source code into model graphs – since the base languages are already well defined, standardised –, it is a never ending battle to follow the changes of the frequently used language extensions, frameworks and non-standard implementation details in these tools.

We intend to delegate the actual graph transformations on the source code level to the most appropriate tools – to the compilers itself, thus we offer a multi-paradigm approach [2]. We call these tools and techniques *customisation techniques*.

4. Customisation Techniques

The customisation uses the same graph manipulations as the model re-generation, but delegates the task to different tools, like C++ compiler and aspect weaver. These tools parse the sources and build up syntax graphs internally and execute the modification requests on these graphs internally.

4.1. Polymorphism

The polymorphism can be exploited in two areas: the design level, where the previously described re-generation algorithm helps its usage, and in the source code generation to preserve implementation level changes. In the first example (see *Fig. 7*) it is used at design level.

From the analysis model there are two sub-models generated: single object view and list view, both for the user interface. In the lists the `showView()` method is polymorphic. If one does not need subclass specific behaviour for *Truck* and *Bus* classes in this method, then the generated *TruckList* and *BusList* classes can be dropped.

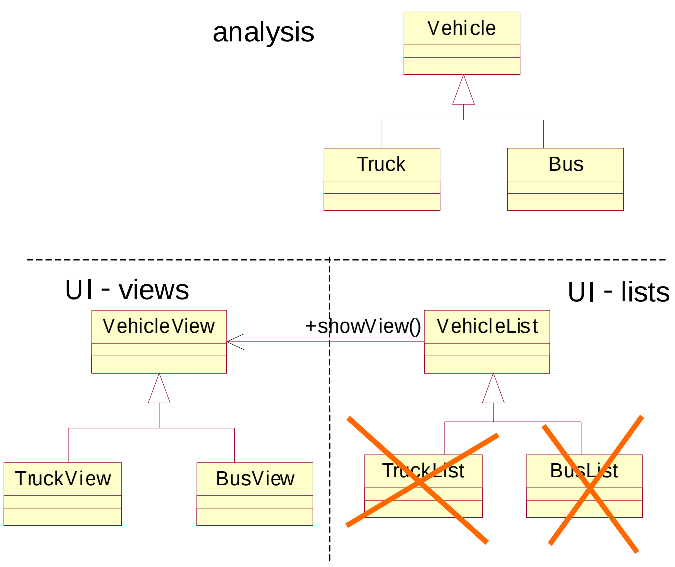

Fig. 7. Customisation via polymorphism - Simplification

The re-generation algorithm preserves this design decision and will prevent the re-insertion of the deleted classes.

The second example demonstrates the usage of inheritance in source code generation as a customisation technique.

Adding or modifying the behaviour of a class can be achieved by subclassing the target class and overriding the necessary method (see *version 1* and *MyCarEditor* in *Fig. 8*).

When the design is modified – the *colour* attribute is added – the change will be populated into the UI design and into the source code as well. Since the modification is separated into a new class which will not be overwritten even if the source is re-generated, the source level changes are preserved.

However, this technique has a drawback: it cannot deal with the case, when a feature or class is deleted from the design. In such cases a compile time error occurs, which will force to implementer to modify the source by hand to follow the changes.

4.2. *Aspect Weaving*

Using an aspect weaver the drawback of inheritance disappears, since an aspect can only be introduced into existing classes. Although it makes the customisation process easier it has the disadvantage of using another tool in the developers' toolcase.

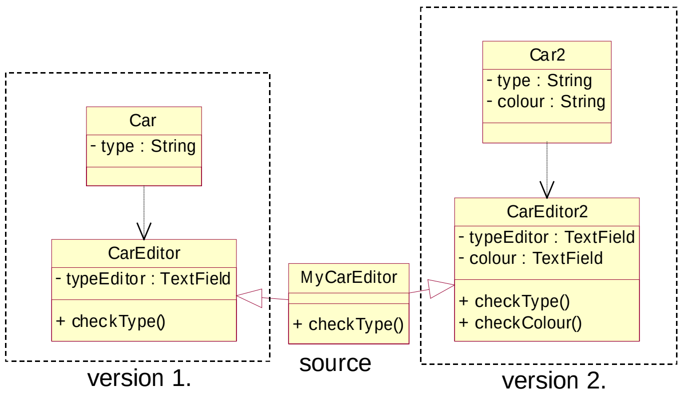

Fig. 8. Customisation via polymorphism - New base class

Using aspects, the graph manipulations happen in the aspect weaver which introduces the code sections from the aspects into the source code.

If the source code is generated from the design model and the aspects are written by a programmer, then this process is basically a way to preserve the hand made modifications.

If the aspects are introduced at the design level, then this is a way to add crosscutting concerns in the original manner of AOP.
In this example a general security policy is introduced, which shall be applied to all descendants of the *Item* class (see *Fig.* 9). The *allowExecution()* and *allowCreation()* methods are inserted before the code sections of the user interface methods. The *allowExecution()* simply throws a security error upon lack of authorization.

4.3. Parametrisation with Meta Information

In the object-oriented world, specialised classes are used created to provide a specific behaviour. This approach provides an excellent way to check the consistency at compile time, but it has some drawbacks:

- Lots of small and trivial classes are created,
- The code tends to be large due to the number of classes,
- Any modification requires the recompilation and installation of the class hierarchy.

For faster development and more flexible architecture, a generic solution can be created. The generic architecture accepts meta information, which is usually restricted to the class descriptions. Based on this information, the basic behaviour can be provided without the need of small and trivial classes.

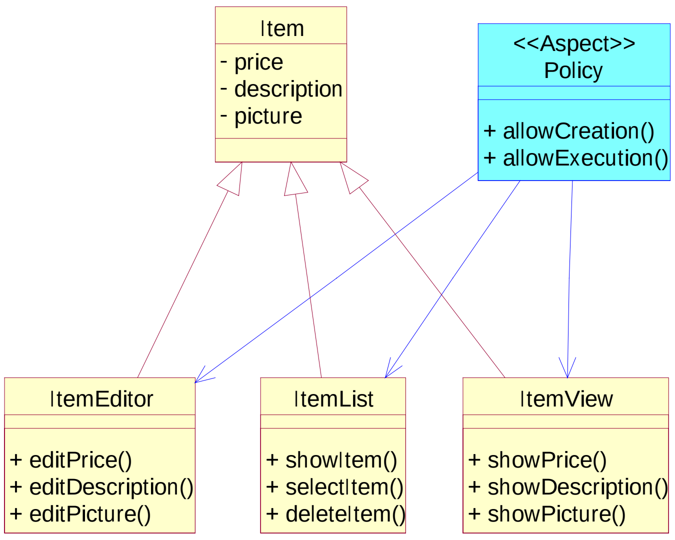

Fig. 9. Customisation via aspects

Actually, the generic architecture generates the basic behaviour on-the-fly; it is done before the compilation in an ordinary environment.[2]

The meta information can be placed on the client on demand, using the installation of a minimal amount of code on the user's computer. It provides the advantage of dynamic code installation without the need of Java-like environment:

+ Minimal amount of code in the client
+ Meta information on demand – minimal amount of information
+ Meta information at execution – dynamic update of the client
– Slower execution
– Lack of compile time consistency checking

Fig. 10 illustrates the class hierarchy for the user interface of a simple *Car* class. The generated hierarchy provides the advantage of compile time checking. The generic hierarchy has a similar structure, but it can be used not only for the *Car* class.

5. Implementation

We chose XMI[16] as the basis of the sub-model and code generating processes. Since XMI is becoming a de-facto standard among the CASE tools, it makes this

[2]see a PropertyEditor based on JavaBean's BeanInfo

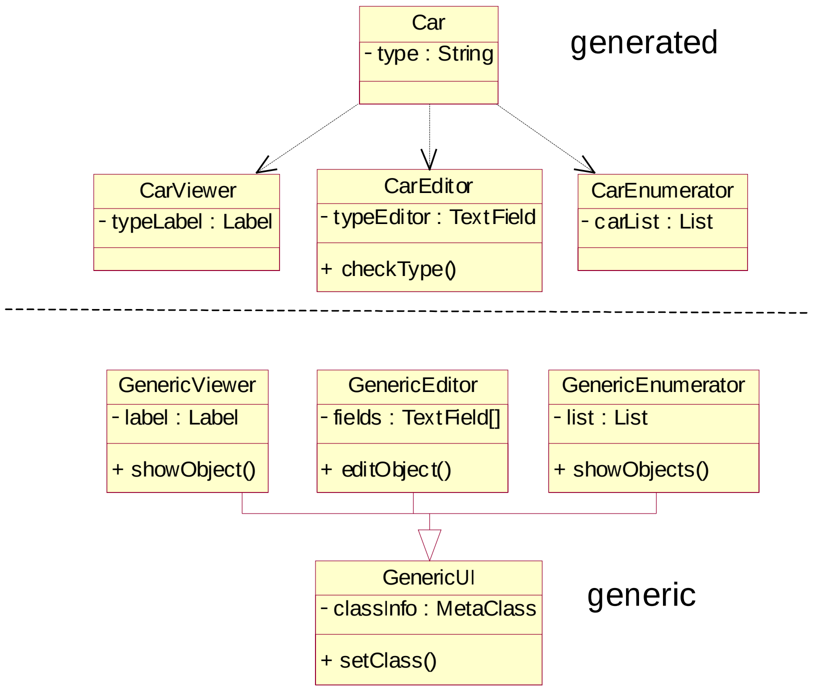

Fig. 10. Generated and generic user interface

set of algorithms and generators inter-operable with other vendors' designer and developer environments.
One possible scenario goes from the analysis model, through the sub-models to a generic user interface or generic database backend.

In this scenario the XMI information may be directly deployed into the generic package. It postpones several consistency checks to run-time, but eliminates the code generation, which can be a desirable speed-up in the early phase of development.

When a mature design is reached the generic behaviour can be turned into static source code based on the same meta information. In the source code more accurate semantic validations and better optimisations can be executed at compile time to produce better quality applications.

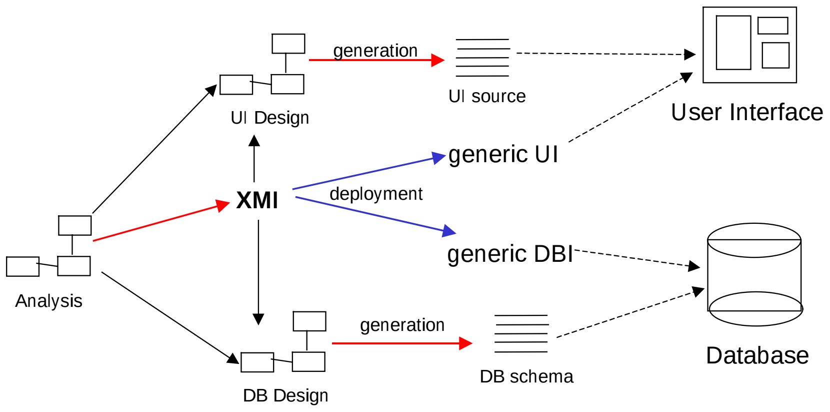

Fig. 11. XMI translation phases

6. Conclusion

CASE tools are currently supporting only specific areas of the development process with limited customisation possibilities.

Our goals were to:

- connect the analysis, design and implementation phases
- provide customisation spots for designers and implementers
- populate modifications in one layer to the whole system

We have achieved these goals by formalising a generic re-generation algorithm to connect the different design layers in three levels of detail, and describing various source-level customisation techniques to extend this algorithm to the implementation's target language.

Although we have achieved these goals we intend to extend our research in the following areas:

- modular generator back-ends for the XMI framework
- complexity metrics for the design model
- generation of behavioural elements from UML state and collaboration diagrams

With our future goals we aim to cover more aspects of the design and provide more easily configurable and predictable tools for the developers.

References

[1] BOOCH, G., *Object-Oriented Analysis and Design*, The Benjamin/ Cummings Publishing Company, 1994.
[2] COPLIEN, J, *Multi Paradigm Design for C++*, Addison-Wesley, 1998, ISBN: 0-201-82467-1.
[3] CZARNECKI, K. – EISENECKER, W. U., *Generative Programming*, Addison-Wesley, 2000, ISBN: 0-210-30977-7.
[4] STROUSTRUP, B., *The C++ Programming Languages Special Ed.*, Addison-Wesley, 2000, ISBN: 0-201-70073-5.
[5] MUSSER, D. R. – ATUL, S., *STL Tutorial and Reference Guide*, Addison-Wesley, 1996, ISBN: 0-201-63398-1.
[6] ALBIR,S. S., *UML in a Nutshell*, O'Reilly, 1998, ISBN: 1-56592-448-7.
[7] FROHNER, Á., An Object Meta Information Framework, *European Conference on Object-Oriented Programming (ECOOP) Workshop* Reader, 1997.
[8] FROHNER, Á., Layered Design Visualisation, *European Conference on Object-Oriented Programming (ECOOP) Workshop* Reader, 1998.
[9] KICZALES, G., Aspect Oriented Programming, *AOP Computing Surveys* 28(es) (1996), p. 154.
[10] KICZALES, G. – LAMPING, J. – MENDHEKAR, A. – MAEDA, C. – LOPES, C. V. – LOINGTIER, J-M. – IRWIN, J., Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland. Springer-Verlag LNCS 1241. June 1997.
[11] LISKOV, B., Data Abstraction and Hierarchy, *SIGPLAN Notices*, **23** No. 5. May, 1988.
[12] RUMBAUGH, J. – BLAHA, M. – PREMERLANI, W. – EDDY, F. – LORENSEN, W., *Object Oriented Modelling and Design.*, Prentice Hall, 1991.
[13] *Unified Modeling Language 1.1*, Rational Software Corporation, 1997. September 1., http://www.rational.com/uml
[14] Xerox, Palo Alto Research Center, *AspectJ Home Page*. http://www.aspectj.org/
[15] Sun Microsystems Inc.: *The Source for Java^{TM} Technology*, http://www.javasoft.com/
[16] XMI at OMG: http://cgi.omg.org/cgi-bin/doc?ad/99-10-13
[17] XMI at IBM: http://www-4.ibm.com/software/ad/standards/xmi.html