Layered Design Visualisation

Ákos Frohner[1]
Eötvös Loránd University, Institute of Informatics, Budapest
Akos.Frohner@elte.hu

Designing an object-oriented system is a process that is well supported by a great number of notations and design techniques such as UML.

Although UML provides notation for almost all aspects of object-oriented software design, it lacks features for describing aspects that are outside of the design domain or require information from different diagrams. For example, there are no good notations for the visualisation of frameworks, friendship relationships, components, meta-level aspects and security considerations.

As a possible solution we propose to use multi-layer diagrams. Such diagrams allow the user of an object-oriented CASE tool to concentrate on the specific feature which she or he is interested in, and filter out the remaining parts.

The basic idea is explained here in more detail through the task of framework documentation. We give further examples using the layered structure to support the design of complex systems and their three-dimensional visualisation.

Keywords: layer; folding; CASE tool; design; documentation; 3D

1. Introduction

The design and documentation of object-oriented systems is well supported by a large number of notations [UMLnot] and CASE tools. Although they are mostly used on computers, they are designed for passive notation that cannot be interacted with.

As the complexity increases in these systems one needs new approaches to handle the large number of elements and to help the understanding of general structures. However, if it is used on a computer screen, the notation can become interactive. One way is to hide the unnecessary details and let the user to focus on a small and comprehensible subset of the components.

We have followed this way in the area of framework documentation and came up with various solutions for the basic problem: hiding the details [Mös97]. This paper presents a possible new solution using layers in the normal diagrams.

---

[1] Currently on leave at the University of Linz, Institute of Practical Computer Science

Section 2 provides the background and the basic concepts for the paper. It also describes how these basic ideas can be applied in an object-oriented CASE tool. Section 3 explains these techniques through a more complicated example in the area of framework documentation. Section 4 gives further ideas of using multi-layer diagrams in the design process and visualising complex systems in 3D. Section 5 shows the current state of the work.

2. Basic Concepts

2.1. Basic UML Notation

In the following examples we use the UML notation, which is described in detail in *[UMLnot]*. The examples are using only a small subset of this notation: classes and various relations between the classes.

Attributes, methods and higher level constructs of the static class diagram are not used, although the same concepts can be applied on them as well.

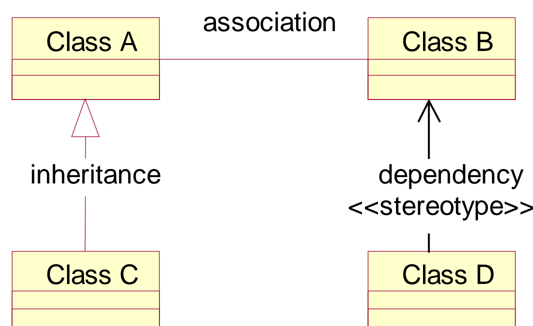

Figure 1. Basic UML Static Class Diagram Notation

In Figure 1 the notation for the classes (rectangular area), simple associations (line without special ends), inheritance relationship (normal line with a triangle shape arrow end) and dependencies (dashed line with arrow end) are introduced. To refine the notation one can add a stereotype with any element ("<<" and ">>" marks).

2.2. Layers in Drawing Tools

In some general drawing tools one can use several drawing layers to create a picture. This way the logical structure of the picture can be preserved: one can work separately on the background and on the figures (see Figure 2), is the same way as an artist draws a cartoon.

During the editing process one can modify only one *selected layer* (or editable layer, marked with "E" in Figure 2), but can choose others and make them visible as well (marked with "V" in Figure 2). It is like laying several transparent foils with the background and furniture figures on top of each other, and editing only the top layer. The author can see the whole scene together, but is not able to modify the figures underneath.

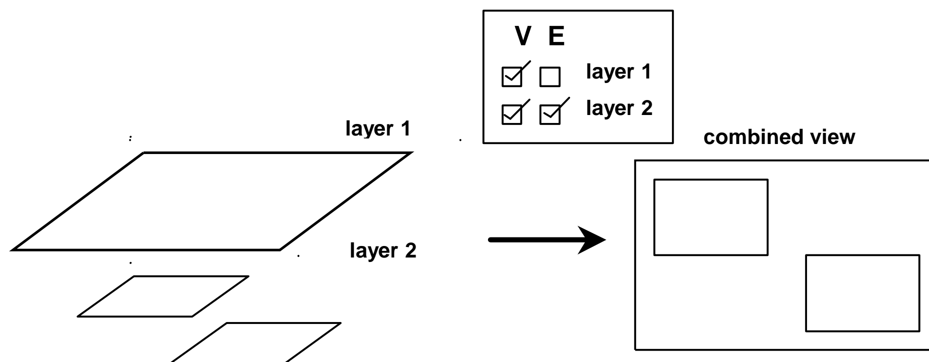

Figure 2. Layers in a Drawing Tool

If the author has finished working with all the layers, she or he can create a final rendering combining the separate layers, although in the original document these layers are still preserved.

One can later return to the original work and continue modifying a selected layer – in which case no other layers will be affected – and create a new rendering if necessary. During the work, layers can be *switched on* or *off* to be included into the working image of the diagram (like adding or removing foils from the working heap).

2.3. Layers

The same layering technique may be applied on an object-oriented CASE tool, as on a general drawing tool. The tool might also provide some additional help to organise the elements, since the model behind the diagram also has an internal structure:

- Inter-layer relations are shown only when both layers containing the endpoints are switched on.
- Type-wise or attribute based selection can automatically place elements to specific layers.

- A simple scripting mechanism can help to maintain the consistency between the layers (see section 4).

The general structure of the tool is the same as in a normal CASE tool.

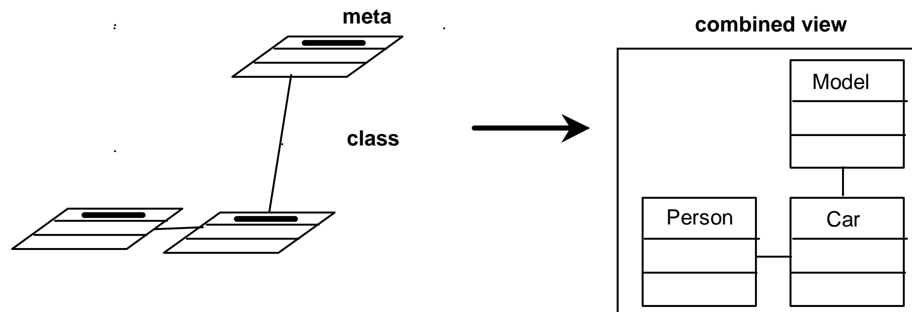

Figure 3. Layers in an Object-Oriented CASE Tool

One can work with the standard elements in a layer: to define new classes and relationships between the classes. The visualisation of the complete design is shown in the *combined view* (see Figure 3), although it can be simplified if one switches off one layer.

For example in Figure 3, if one layer is switched off, then the *Car* or the *Model* class will not be visible, therefore the relationship between them will be hidden as well.

2.4. Folding Elements[2]

Based on the object-oriented model itself, a tool can provide more sophisticated manipulations on diagrams and not only for the authors, but for the readers as well.

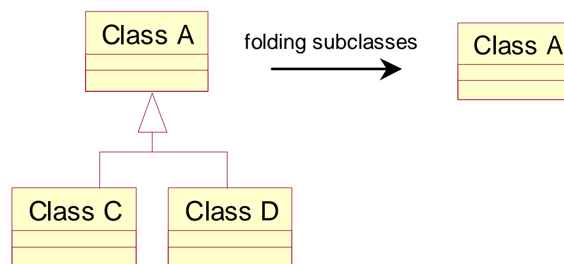

Figure 4. Folding in an Object-Oriented CASE Tool

[2] The work of Stefan Chiettini (University of Linz, Institute of Practical Computer Science) may provide more information on this topic.

Details of the diagram can be shown or hidden on demand. Such details are the attributes and methods of a class or a branch of subclasses in an inheritance tree (see Figure 4).

The major benefit of this approach is the convenience in the usage: the user does not have to place the unnecessary elements (the branch of subclasses in this example) into a separate layer, because the tool can make this selection.

2.5. Small Comparison

However, automatic folding might constrain the usage in cases where

- the user wants to hide other classes of the model as well or
- wants to hide all the subclasses except one.

The layered approach solves these problems with the cost of manual selection, thus the ideal solution would be the combination of both techniques.

The main difference of the presentation between folding and layers technique is in the positioning of the elements. Folding usually rearranges the elements whereas they are left at the same position in layers.

Rearranging the elements might provide a more compact view of the investigated part, thus reducing the size of the viewed diagram. The cost of this compactness lays in the implementation, where sophisticated automatic layout techniques are required.

Contrary to folding the elements are left at the same position in layers, thus they can be found more easily. The implementation should also be simple by adding the layer number to each element.

3. Examples

The following examples are to demonstrate the usefulness of the above-described basic techniques. For simplicity only static class diagrams are drawn, but the same approach can be taken to draw the dynamic ones.

3.1. Simple Filtering

The simplest usage for layers would be to filter the final view of a static class diagram.

The “complex” model on Figure 5 can be simplified by viewing only one layer of the design, for example the unnecessary subclasses can be eliminated (see Figure 8).

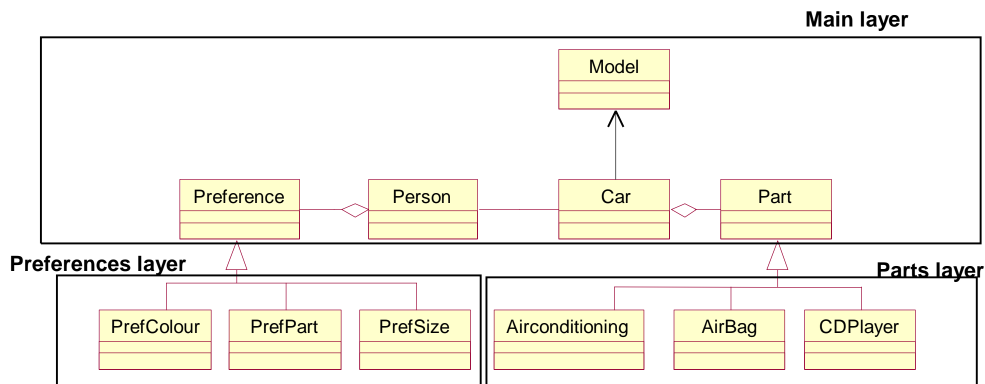

Figure 5. Unfiltered Class Structure

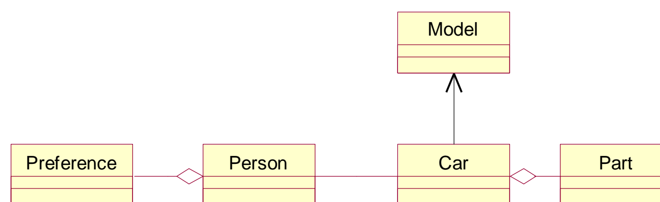

Figure 6. Filtered Class Structure

One could achieve the same effect with folding the subclasses of a non-leaf class, although folding of unrelated – without one common superclass – classes is hard to specify. In this example the subclasses of *Preference* and *Part* classes can be placed on one layer and eliminated from the final view by disabling that one layer.

3.2. Framework Documentation

A framework usually has some *hot-spot* (e.g. which class to subclass, which method to override and which event to catch), where the functionality can be customised or extended. These are rarely located only in one class (e.g. tracking a specific event might involve several methods), thus automatic hiding of irrelevant parts is not possible.

In Figure 7 an extendable part of a CASE tool is shown. The *ModelElement* class is the root of the classes to store the meta information of a class diagram (e.g. classes, attributes, methods, parameters). The *ViewElement* is a basic class in the user interface, which knows how to display itself. There are two interfaces in this small example representing the event flow from the user interface (*GeometryListener*) and from the meta-model (*ModelListener*).

The abstract class *ModelElementView* is a basic solution to display meta-information pieces. It can be customised to represent the various elements of a model diagram. It provides the basic infrastructure for the framework user that she or he will have to provide just the minimal amount of code to display various elements.

Class *ClassView* shows how to extend *ModelElementView*. To understand the functionality of the framework, the bodies of some methods are also shown in pseudo code.

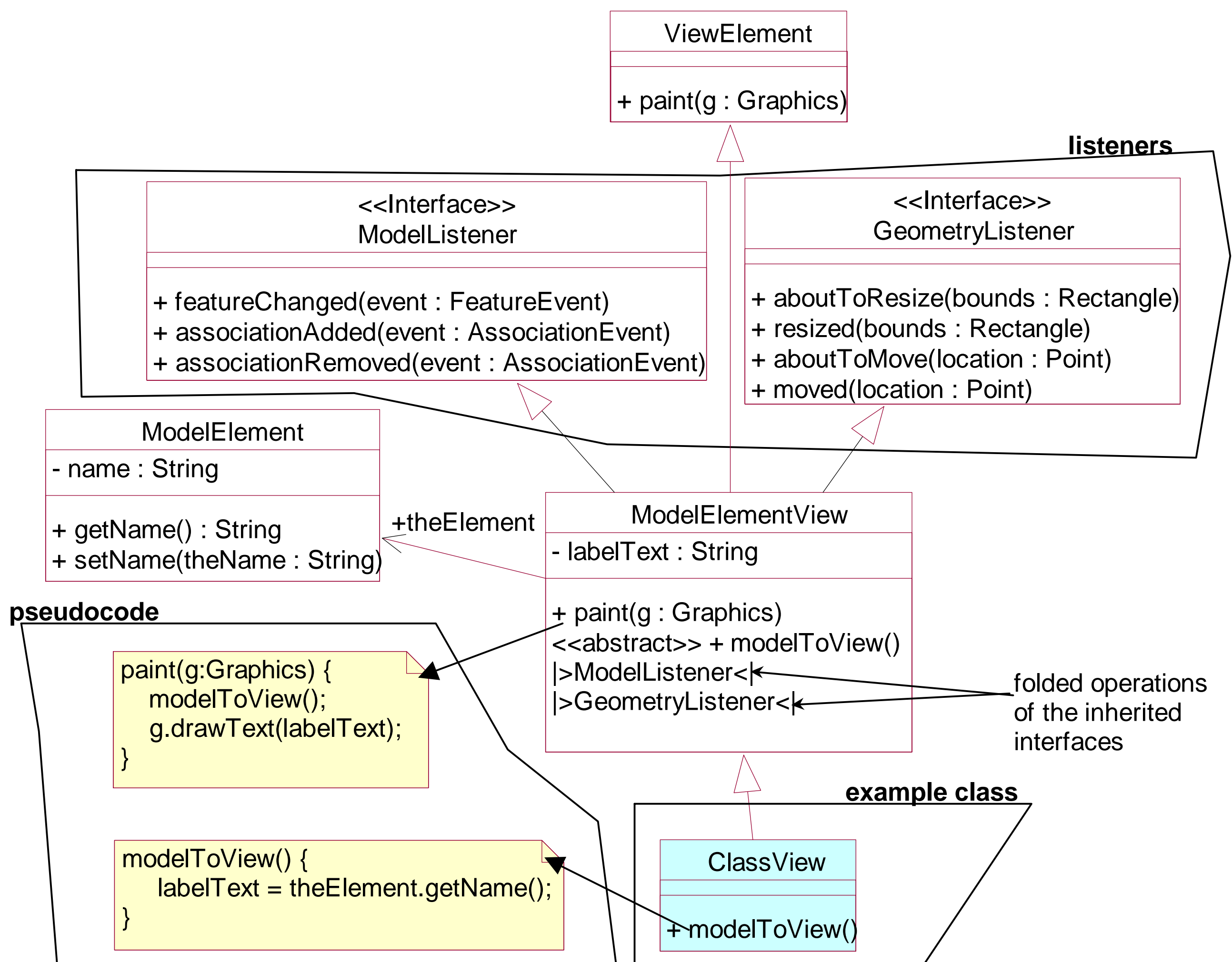

Figure 7. Framework Usage Example

The layers and folding can be used to hide some parts of this diagram, and thus to simplify the understanding of how to extend the *ModelElementView* class. One could hide the *listeners* and the *pseudocode* layer and fold the methods implementing the two interfaces. In this way the number of displayed boxes are reduced to four. If the user understood the basic structure of the framework, she or he can incrementally fold out the compressed parts and visualise the hidden layers.

One can also define a recommended sequence of layers, and thus a sequence of steps in which the diagram should be read. Such a sequence (a film) can even be automated (see [Mös97]).

In an active CASE tool one might even hide the example class and try to design ones own extension instead of the *ClassView* class. Since this work will be done on a new layer, it would not affect the underlying example (probably set to read-only), but could have the advantage of seeing all the necessary information inside this workplace.

The diagram creation process could be simplified by providing tools to fold parts of the diagram and later associate these parts to selected layers.

4. Further directions

This section expands the multi-layer idea to the active design process and to the visualisation of general structures in three dimensions.

4.1. Designing Complex Models

There can be complex interdependencies between the classes in large models. This example demonstrates how can multi-layered diagrams help their visualisation and adds some further ideas to keep unusual dependencies consistent in the model. The key idea is to map small diagrams on each other instead of lay them out side-by-side.

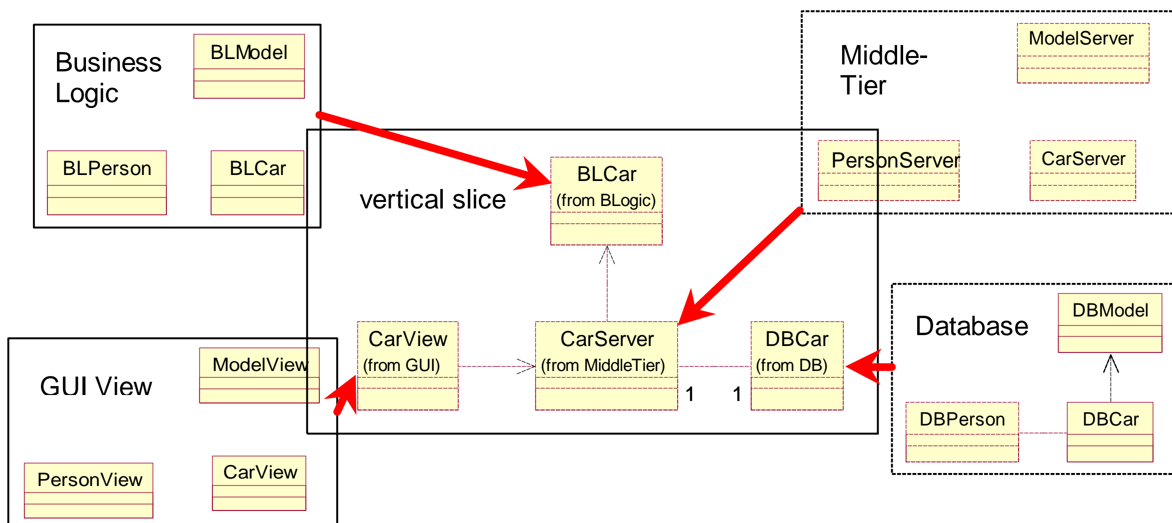

Figure 8. Derived Classes in Several Packages

Unfortunately, after the initial designs phase, the consistency is no longer maintained between the derived classes. Although there is already some software tools to reverse engineer source code and provide consistency between implementation and design, but there are no widely used tools to provide similar synchronisation between object-oriented design and analysis phases.

The dependency between the original and the derived classes is not inheritance (a business logic class might not contain any attribute, and the database class might not contain any method), but some kind of evolutional relation. A tool could – and should – provide some kind of scripting support to generate the first version of these derived classes and might provide further support to maintain this dependency, but the *designer needs* some kind of *visual feedback* as well. If she or he modifies the original class, it is highly probable that the derived classes have to be modified as well.

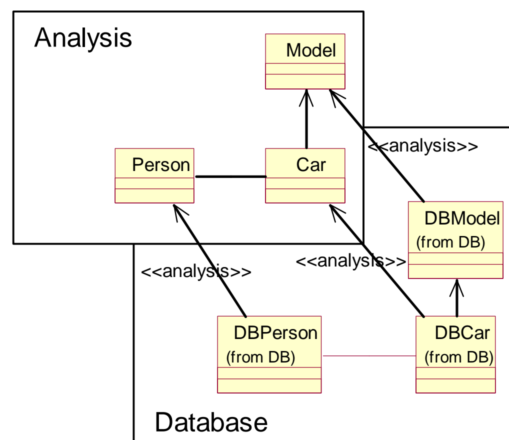

Figure 9. Dependency between Analysis and Design classes

It is clear that the complexity of the whole system is relatively big (see Figure 8 or Figure 10) compared to the number of original classes (three). One can not express all the relations, because the resulting diagram would be quite crowded.

If only two of these class sets are shown at the same time (see Figure 9), then the simplified diagram is more acceptable for easy understanding. The **layered** approach might help, by enabling the designer to dynamically switch on and off specific layers and create different renderings of the same design on demand.

In this example, when the *Person* class is created, the designer should add the *DBPerson* class to the *Database* package and the *PersonView* class to the *GUI*. The missing pieces can be easier noticed if complex class diagrams are mapped on each

other using layering (see Figure 9) instead of placing them side-by-side in separate diagrams like in Figure 8.

Another possibility is to **fold** the unnecessary elements to simplify the diagram. The problem with folding is the selection of the elements: folding is usually applied to a logically or geometrically related part of the diagram, although one might want to handle classes together which are hardly described by any of these relations.

In this example a tool can not select all the classes from user interface packages or select the classes related to the class *Car* (see Figure 8: *vertical* slice) in one operation to fold them. However, if the designer can select multiple unrelated parts of the diagram for folding, then the same goal can be reached using this alternative approach, as with the previously described layering.

4.2. Using the 3rd Dimension

Using layers in an object-oriented design tool simplifies the individual diagrams, but the user still has to make a big step if she or he wants to understand the whole structure.

Two-dimensional diagrams, containing the overall design, are likely to get overcrowded after a certain point, because there are either

- too many classes or
- too many relations (connecting lines crossing over)

on one sheet. The only possibility to simplify the understanding of too many classes is to organise them in a better structure. However a better organisation might ravel the web of relation lines as well, there is another solution for this problem – borrowed from the hardware design–: step out to the 3rd dimension!

If a designer uses the layers to organise tightly coupled classes, then it already provides a natural structure to visualise the whole design in three-dimension as well (see [FeJo98] on other three-dimensional layout possibilities).

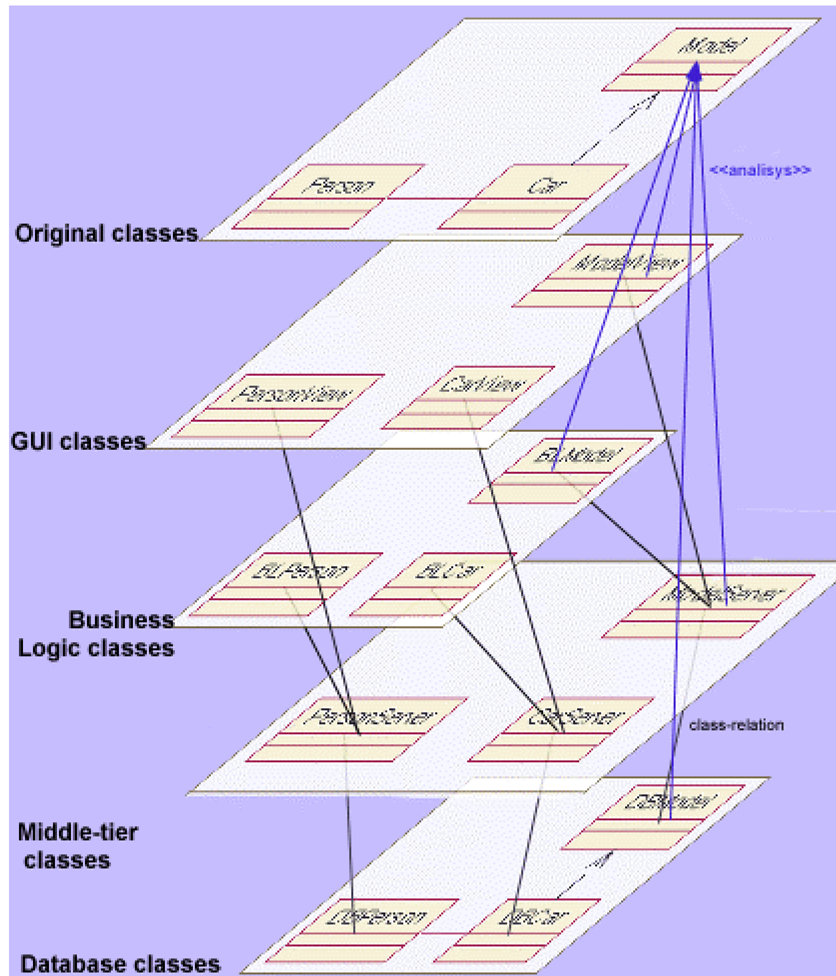

Figure 10. Layers in Three Dimensions

Figure 10 demonstrates this idea on the example of section 4.1: the layout of the static class diagrams is preserved in the layers of the three dimensional view, thus the designer can refer to the original diagrams for further details.

The inter-layer relations are drawn as lines in the three-dimensional space, so they are not likely to cross each other. Although the rendered image might still be fuzzy, choosing a more convenient viewpoint could unveil its structure in a better way.

5. Current State of the Work

The idea, presented in this paper, is yet in conceptual stage, but we intend to build a prototype implementation to explore its usefulness in practice.

Unfortunately we do not have access to the source code of an existing object-oriented CASE tool, therefore we need to build a simple one ourselves. The application is partially based on an earlier work on meta information framework [Fro98] (the model behind the diagrams) and on prototype documentation tools,

which were built during the co-operation with the Institute of Practical Computer Science at the University of Linz.

In the first stage the tool will support static class diagrams, since source code analysers can easily support their computer aided generation. It will also allow free style drawing using graphical primitives, such as lines, boxes and textual notes, to enrich the functionality. Later we intend to add support for the dynamic behaviour as well.

The tool will follow the data storage concept of the above-mentioned meta information framework. It will not only store the current state, but also allow the storage of manipulations with the help of action objects. This mechanism enables unlimited undo steps and the recording of manipulation sequences, which can be later reviewed or replayed as a film.

At the moment we left the implementation of the ideas in section 4 as optional.

6. Conclusion

The layering technique adds some new notational feature to the existing possibilities of UML, but the main impact is on the design work itself. Using layers to associate elements allows users to *express their own way of thinking* above the logical structure of the model (i.e. package boundaries).

The layering idea adds only one new attribute (i.e. the layer number) to the elements of a diagram, therefore the handling of layers (selecting and switching them on and off) and the presentation of the enabled layers, should impose only a small overhead in CASE tools. The rendering of a filtered view should also be easy to implement.

References

[BRJ99] Booch Grady, Rumbaugh James, Jacobson Ivar:The Unified Modeling Language User Guide, Addison-Wesley Longman, Inc. 1999.

[FeJo98] Feijs Loe, Roel de Jong. 3D Visulaization of Software Architectures, Communications of the ACM, December 1998/Vol. 41, No 12, p73-78.

[Fow97] Fowler M., Scott K.; UML Distilled, Addison-Wesley Longman, Inc. 1997.

[Fro98] Frohner Á.: Framework Design and Documentation, ECOOP'98 Workshop Reader, LNCS 1543, Springer-Verlag 1998, p5-6.

[GHJV95] Gamma, et al.: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Inc. 1995.

[KoM95] Koskimies K., Mössenböck H.: Designing a Framework by Stepwise Generalisation. Proceedings of the 5th European Software Engineering Conference (ESEC'95), Barcelona, LNCS 989, Springer-Verlag 1995.

[Mös97] Mössenböck H.: Films as Graphical Comments in the Source Code of Programs. TOOLS USA'97, Technology of Object-Oriented Languages and Systems, Santa Barbara, July 1997.

[UMLnot] UML Notation, version 1.1, 1st of September 1997, UML 1.1 documentation set.

[UMLsem] UML Semantics, version 1.1, 1st of September 1997, UML 1.1 documentation set.