Structuring and Refinement of Class Diagrams*

Klaus Bergner, Andreas Rausch,
Marc Sihling, Alexander Vilbig

a1intern@in.tum.de
Phone: +49-89-289-22685
Fax: +49-89-289-25310

Institut für Informatik
Technische Universität München
D-80290 München, Germany

1st June 1998

Abstract

The class diagram notation of current graphical object-oriented modeling languages does not scale well with the increasing complexity of modern systems. To overcome this weakness, we propose the concept of structured interfaces, a flexible refinement relationship between class diagrams and the use of attribute, operation and relation templates to abstract from implementation details. The usefulness of the introduced concepts is illustrated by the development case study of a web-based distributed editor. Finally, we present suggestions for the integration of the presented structuring and refinement concepts into CASE tools and documentation systems.

Keywords: Object-Oriented Software Engineering, Modeling, Class Diagram, Refinement, Attribute, Operation, Relation, Structured Interface, CASE, Internet Application

1 Introduction

It is a common observation that size and complexity of software systems are increasing steadily. This is partly due to the integration with other systems over local and global networks or even over the internet: Connecting existing systems generally increases the number of business-oriented and technical interfaces that have to be considered during development. Additionally, the necessary support for complex technical mechanisms also raises the number of the features of these interfaces and their corresponding implementation classes.

To develop and understand highly complex distributed object-oriented systems, various modeling languages have been proposed. The most popular example is the unified modeling language UML [Gro97] which combines features from many earlier approaches, like OMT [RBP+91] and the Booch method [Boo94]. One of the central modeling techniques of UML are so-called *class diagrams*: They show the classes of a system, their attributes and operations, and the relationships and dependencies between them. Class diagrams are used early in the development process to model the issues of the system with respect to the application domain in question. Later, during the transition from analysis to design, they are refined and enriched with additional classes, attributes, and operations. Finally, tools are used to generate code skeletons from the most detailed class diagrams. The actual functionality may then be implemented by the developer.

*This paper originated in the FORSOFT project A1 on "Component-Based Software Engineering", which is supported by Siemens ZT.

This illustrates the importance of class diagrams as the basis of communication during all phases of object-oriented system development. Usually, development documentation includes a series of connected and increasingly complex class diagrams which differ significantly in their level of abstraction. Therefore, it is necessary to consider the following two key aspects of development with class diagrams:

Structuring of individual class diagrams at a given level of abstraction is important to understand and manage the contained development information. Unstructured and overly complex class diagrams hinder the communication about the system.

Structuring of a set of class diagrams at different levels of abstraction is necessary to trace the progressing system development from analysis to implementation. A flexible concept of refinement is used to connect the individual class diagrams and their contained model elements from a logical point of view.

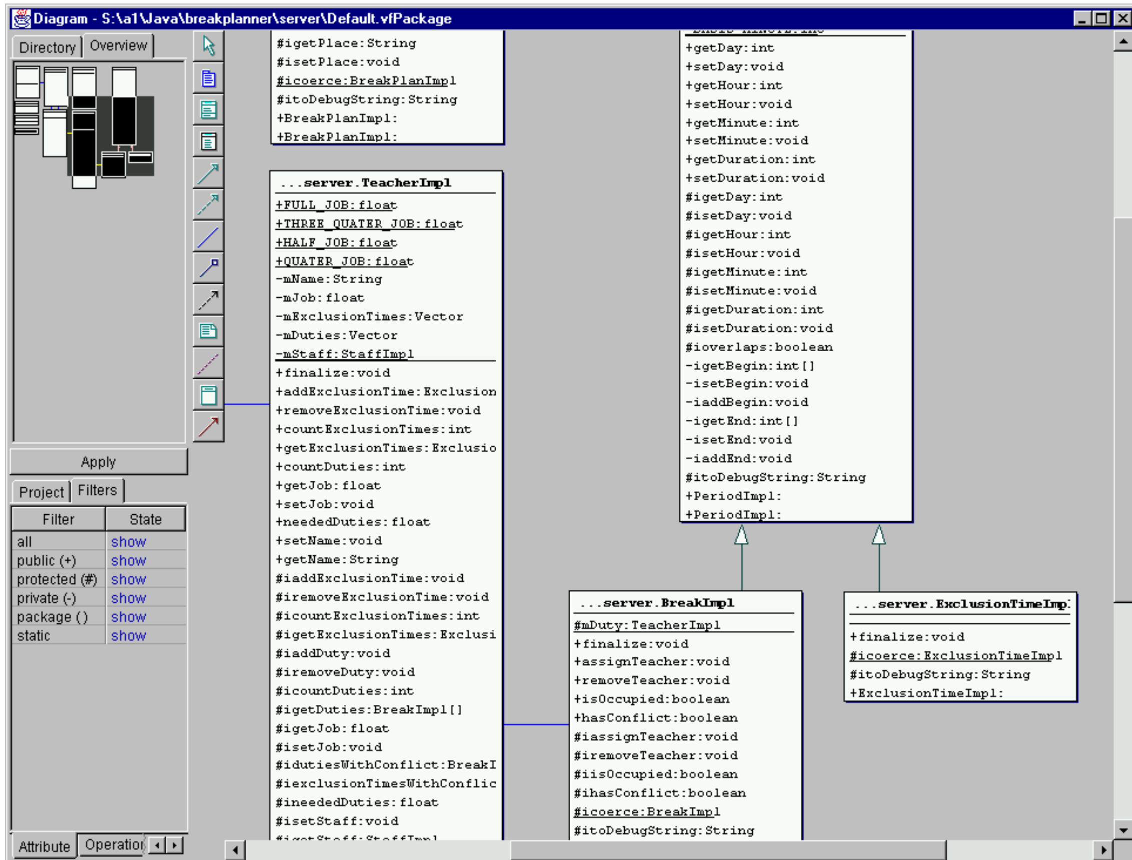

Figure 1: Screen of TogetherJ with Break Planner Project

Current development methods and modeling languages do not support these two areas very well. Although UML, for example, provides the concept of packages to group and manage entire sets of classes, there is currently no means to structure class diagrams on the basis of individual classes and their interfaces. The majority of available tools only provide a rather primitive filter mechanism for the attributes and operations of a class. It is usually based on characteristics like access level—a typical example is to display only the `public` features of classes. Class diagrams with complete interface information usually get very large and rather incomprehensible, as can be seen from Figure 1, a screen shot of the tool TogetherJ [Obj98]. The right side of the figure shows a part of a medium-sized class diagram. In the lower left side, all features of the interfaces are displayed. The upper left side provides an overview of the whole class diagram, the dark grey area corresponding to the part of the diagram displayed to the right.

In Section 3, we present a flexible approach for grouping features of individual classes in order to improve understanding and management of such complex class diagrams.

As mentioned previously, it is also necessary to structure class diagrams with respect to the development process itself. Changing user requirements or changing technical infrastructures result in constant evolution of the corresponding class diagrams. Therefore, the ability to trace requirements during all phases of development as well as documentation of design and implementation rationales becomes increasingly important.

UML offers the concept of *refinement* between two descriptions at different levels of abstraction. For example, an implementation class with additional attributes and operations may refine a corresponding design class. However, the proposed concept of refinement is not comprehensive and flexible enough [BRS98b, BHH$^+$97]. For example, it is not possible to specify that a given attribute is refined by an entirely new class or that a given analysis class is dropped during design because the corresponding features will be realized outside the system.

We, therefore, introduce an extended concept for refinement in Section 3.2. It allows the tracing of requirements and the documentation of design rationales by refining class diagrams. Refinement relationships can be specified and visualized for individual elements of the model, like attributes and operations of a class, for example. Additionally introduced development techniques like attribute and relation templates, support refinement by abstracting from the details of low-level design and implementation.

Thereby, refinement is used to structure a set of class diagrams across different levels of abstraction, whereas grouping of attributes and operations is used to structure a single class diagram on a given level of abstraction. Both concepts are illustrated in the context of an example internet application that is described in Section 2. Section 4 considers the tool support which is necessary to implement both concepts during practical development. Finally, Section 5 concludes with a short summary and evaluation of the proposed concepts.

2 Application Example

Our example application is a typical distributed client/server system with many thin clients and a central server. The customer requirements for this application originate from the DACH group [DAC] and are based on a real-life scenario [RSLML96].

The customer specification considers the following application scenario: Teachers have to supervise pupils in the various parts of a school building during the breaks. The assignment of teachers to breaks is specified in the break plan of the respective building part. Each break must be supervised by a teacher, and teachers are assigned to breaks depending on the time they spend for teaching—a full-time teacher has to supervise more breaks than a teacher with only a couple of lessons per week. Teachers may provide the school with time periods during which they cannot be assigned to breaks because of other duties.

The users of the system are responsible for maintaining the break plans and the teaching staff data (the respective persons are called "plan editors" and "staff editors"). The system allows the user for example to create and to delete break plans, to assign teachers to breaks, and to manage a list of the school's teachers. Additionally, the tool computes some statistical values for plan editors, for example the number of breaks a teacher still needs to be assigned to.

One of the requirements stated during requirements analysis was that plan editors may "work at home over the internet with a Java-capable browser". The distribution architecture is restricted considerably by this requirement because it implies that GUI objects are managed by applets running on client computers. We do not consider form-based GUIs because they can not provide the look-&-feel required by the customer specification.

The requirement also implies the existence of at least one server with a "real" Java application that can handle persistent information—applets running on browsers are usually forbidden to access local files according to the sandbox safety model of Java [Jav97]. Hence, we chose a client/server architecture. All the data pertaining to objects, like teachers, breaks, and so on, is stored on the central server. The server also performs the functionality of these objects and takes care of the enforcement of the consistency conditions.

Figure 2 shows the GUI of the Break Planner application. It is realized as a Java applet that can be downloaded from the server by a Java-enabled web browser and then accesses the server's data and functionality via the Java RMI communication mechanism [SUN97b].

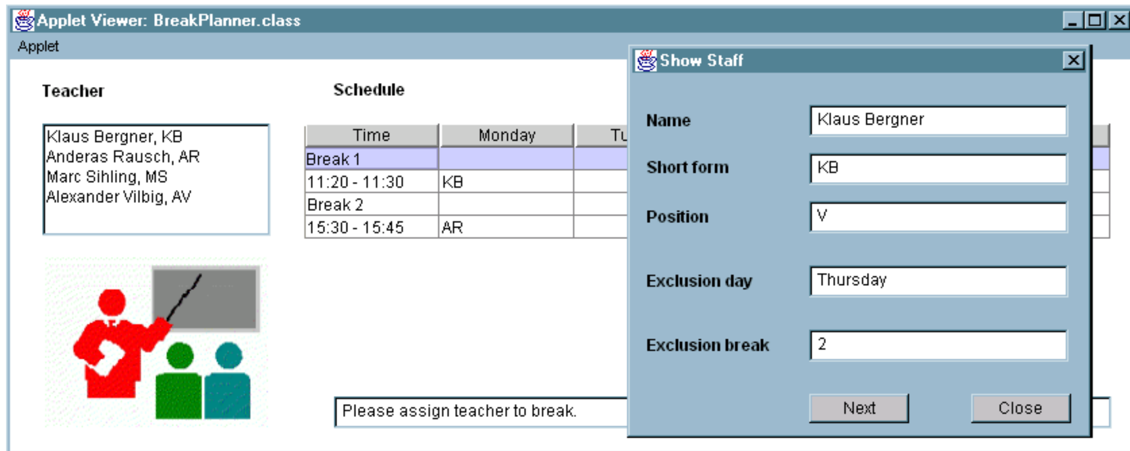

Figure 2: Break Planner Client GUI

Although the Break Planner application is rather small compared to most client/server applications, all the inherent complexities and intricacies of such systems apply. Its main interfaces and mechanisms, namely, the application and data server interface, the GUI client, and the communication infrastructure, may be found in similar form also in larger systems. A full description of the whole development process can be found in [BRS97].

3 Structured Class Diagrams

As mentioned above, there are two key aspects related to working with class diagrams:

1. A means of structuring the numerous features of complex classes is necessary to improve understanding of their various capabilities.
2. A flexible concept of refinement is needed to support traceability of requirements and documentation of design and implementation rationales.

In Section 3.1, we propose a simple, yet effective approach for grouping the features of a class within a given class diagram by assigning them to logical categories that may be hierarchically structured. Even in the context of relatively small class diagrams like the ones of the Break Planner, grouping allows to comprehend the structure of classes with many features much faster. The use of well-known commonly named responsibilities in different classes could help to understand the different capabilities of the entire class more easily.

To support flexible refinement, we introduce in Section 3.2 the notion of a *Flexible Refinement Relationship* between any parts of the specification or design model. Additionally, we propose to use *Attribute and Relation Templates* in order to abstract from the details of low-level design and implementation. Together, these concepts allow to trace requirements across several logically connected class diagrams while omitting unnecessary details at a given level of abstraction.

We apply the proposed concepts in the context of the internet application described in Section 2. This example-oriented approach helps to illustrate the basic ideas and the potential benefits of our approach.

3.1 Grouping Attributes or Operations

As shown in Figure 2, the Break Planner GUI employs the class `JTable` of the Java Foundation Classes [SUN98] to display the current schedule. Before using such a powerful GUI component,

however, it is necessary to understand its interface. `JTable` exposes a very extensive interface with 25 different attributes, 7 constructor operations and approximately 125 operations (see Figure 3).

Figure 3: Part of the Original Interface of JTable

It is obvious that a simple assignment of this large number of attributes and operations to logical categories would strongly enhance understanding and use of this interface. This especially important in the context of visual programming if certain features of interfaces are to be connected with an according tool.

The features of `JTable`, for example, may be naturally grouped in display, interaction and model-related categories. Display-related attributes may be further structured into layout and appearance-related categories whereas interaction-related attributes may be distinguished for being selection or editing-related. These introduced categories lead to a clearly structured interface as visualized in Figure 4 that is a lot easier to understand.

Note that the applied grouping need not be disjunct—some attributes or operations may belong to several categories depending on the chosen structure. The attribute selectionBackground, for example, refers to a colored background and is, therefore, considered as appearance-related. On the other hand, it is also relevant to the handling of selections and is consequently assigned to the category 'Selection' as well (see Figure 4). This fact will not irritate the user of the interface as he is able to study it from different point of views with the aid of a given logical structure.

During design and modification of the interface, however, it is necessary to ensure consistency across the overall chosen structure with respect to name, type and signature of attributes and operations. This has to be strongly supported by adequate tools as described in Section 4.

Although the chosen structure of a given interface is arbitrary, it may prove beneficial to require a set of components to adhere to the same structure. This applies in particular to class libraries, like the *Java Foundation Classes* [SUN98], with a large number of similar classes. The understanding of the interface and its structure belonging to a given class, like JTable for example, thus facilitates the understanding of the other classes, like JTree or JList.

It is furthermore conceivable to define a number of alternative structures for the same interface.

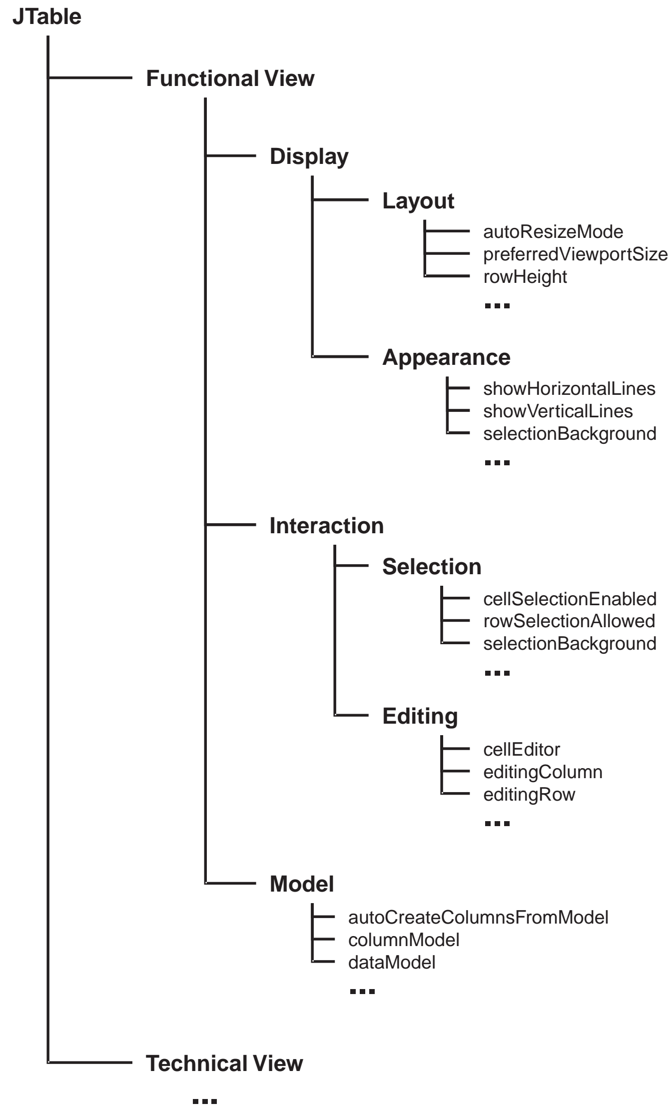

Figure 4: Part of the Structured Interface of JTable

Each structure represents a different logical view on the classes of a class diagram and may be used by different persons during the development process. A system architect, for example, is responsible for integrating the various classes of the system and will, therefore, mainly distinguish between communication features and functional features. A GUI designer, on the other hand, is more interested in display-related features and might therefore employ a similar structure as described above. This approach leads to several possible views on the same structured interface as denoted by the terms 'Functional View' and 'Technical View' in Figure 4. Again, we rely on tools to support the comfortable definition and presentation of these views.

Note that the proposed grouping of attributes and operations implies no semantics of its own, i.e. it is possible to resolve the stucture by generating a "flat" interface without affecting either implementation or the behavior described by the interface. This is in contrast to the more advanced concepts that are introduced in the following sections.

3.2 Refinement of Class Diagrams

The design phase of the development process is mainly concerned with constructing a more detailed, refined and implementation-oriented class diagram starting out from analysis classes. Focusing on the classes of the system during design facilitates the transition to the final implementation as classes are the prevalent means of structuring object-oriented program code.

Creating a design model out of the analysis model involves the following development actions:

1. Some analysis classes may be taken unchanged to design.
2. Some analysis classes may be dropped because they denote concepts not implemented by classes in the final system.
3. Some analysis classes, attributes, operations, and associations are merged together or split up, according to various reasons, like, for example to optimize access paths, to cache data for safety reasons, or to refine very complex analysis classes.
4. New classes, attributes, operations, and associations necessary for modeling technical concepts of the intended implementation may be introduced.

In our opinion, the design process is seperated in two phases, the first of which is focusing on evolving a business-oriented class diagram and thus neglecting technical requirements. The designer most likely uses the first three refinement steps which we support and document by a general, flexible refinement relationship in the following section. The fourth operation is performed in the last design phase to cast the evolved structure into a technical infrastructure considering necessary implementation details. We propose so-called attribute and relation templates to perform this refinement while still retaining a certain level of abstraction.

3.2.1 Flexible Refinement Relationship

UML already defines a notion of refinement as a relationship between two descriptions of the same element at different levels of abstraction. However, it is not possible to illustrate the refinement of an operation of a given class to a set of operations spread over several different classes or the refinement of a set of classes to a realization outside the software system.

In case of the Break Planner example application, the analysis class diagram as shown in Figure 5 includes the class Statistics, which is responsible to maintain statistical data about the breaks each teacher has to supervise as well as the individual share of the total workload.

Figure 6 illustrates the corresponding business-oriented class diagram and its refinement relations to the analysis class diagram (using steps one to three introduced in the last section). During the transformation from analysis to design, the class Account has been dropped, and the functionality of the class Statistics has been distributed among other classes.

Account is not retained in the design class diagram because the intended implementation platform already provides suitable account and authorization mechanisms, e.g. by http logins or file access modes. As a consequence, the implementation of an additional account mechanism is unnecessary, and the class Account now falls outside the system boundary.

Statistics was also not adopted because it represents more of a collection of special-purpose functions than a “normal” class: It has no attributes, it does not participate in non-trivial associations and is thus not used to hold data. Instead, the class computes certain values from the attributes of other classes. During design, the handling of such special-purpose functionality is a common problem[1].

The notation introduced in Figure 6 is an adaption of UML description techniques. As proposed by UML, refinement is shown as a dotted arrow from the refined to the abstract class. In our proposed concept of refinement, however, we allow refinement relations between any kind of model element, e.g. classes, attributes, or operations. The graphical notation denotes this relationship by a dotted arrow from the refined to the abstract element. If a class or another element is dropped altogether, we illustrate this by a dotted line surrounding the element in question, as shown for Account and Statistics in Figure 6.

[1]Further information about this problem and possible solutions may be found in [BRS97]

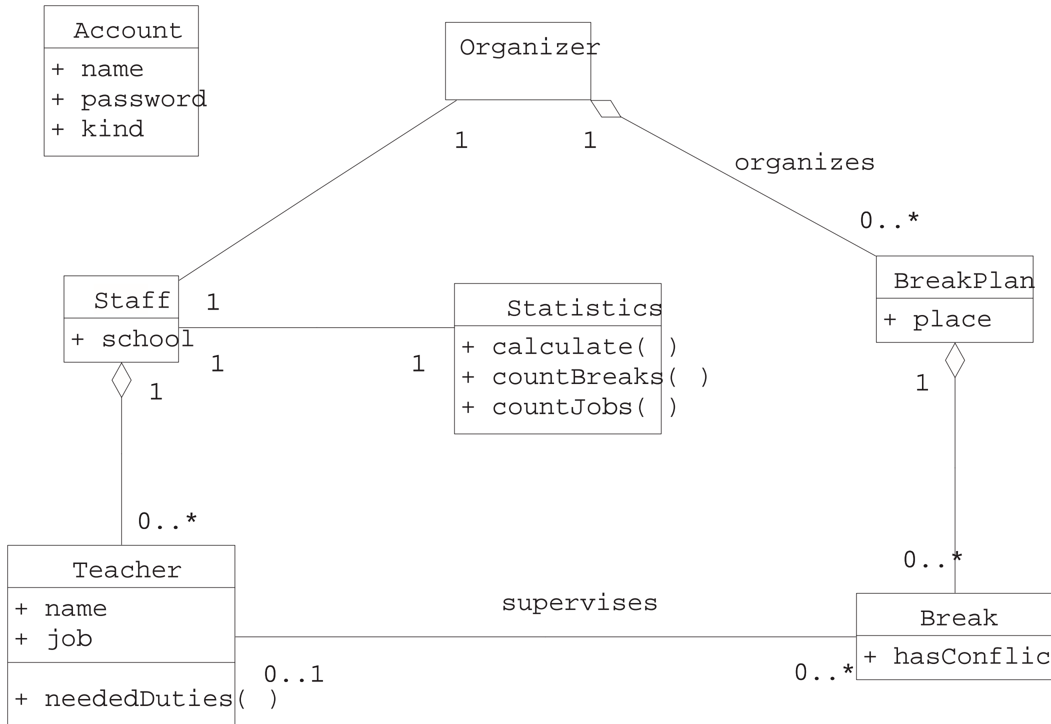

Figure 5: Analysis Class Diagram of the Break Planner Application

A label may be attached to a refinement to document its kind. Moreover, the corresponding design or implementation rationale can be specified this way. An example is the refinement of the calculate operation from the analysis class Statistics. It was splitted up into two new operations one of which needs to be executed on the server (see Figure 6).

Refinement relations specify dependencies between elements and thus can be used to trace the elements affected by possible changes of initial requirements. Needed modifications can now be easily distributed over the refinement of the respective classes.

3.2.2 Attribute, Operation, and Relation Templates

The last step in the refinement of a system model is the translation of the business-oriented class diagram to code which involves specification of implementation details as well as aspects of the technical infrastructure in which the system is to be embedded. If the model provides enough information, this step can at least partially be automated and thus taken from the user's responsibility (cf. Section 4). In this section, we introduce so-called *templates* to specify the proper translation of related model elements to low-level design and implementation. This leads to an abstraction from the involved details and thus helps to keep the class diagrams "tidy".

Conceptually, a template may be defined as a separate unit of structured development information that may be logically connected to elements of the class model, like attributes, operations or relations. It is used to specify the way a given model element is to be translated during low-level design and implementation. This allows the use of more abstract model elements at earlier phases of the development process.

As example to illustrate this idea, consider the refinement of attributes: In the analysis class diagram of the Break Planner application as shown in Figure 5, the class Teacher contains the attribute job to store the teacher's workload. During design, the developer is likely to specify the appropriate circumstances under which this attribute may be accessed by other classes. In our example, the designer decides to encapsulate the access to the attribute `job` using two operations `getJob()` and `setJob()`. As he made this decision at a rather abstract design level, the corresponding operations are present in the class diagrams up to the implementation phase during which they

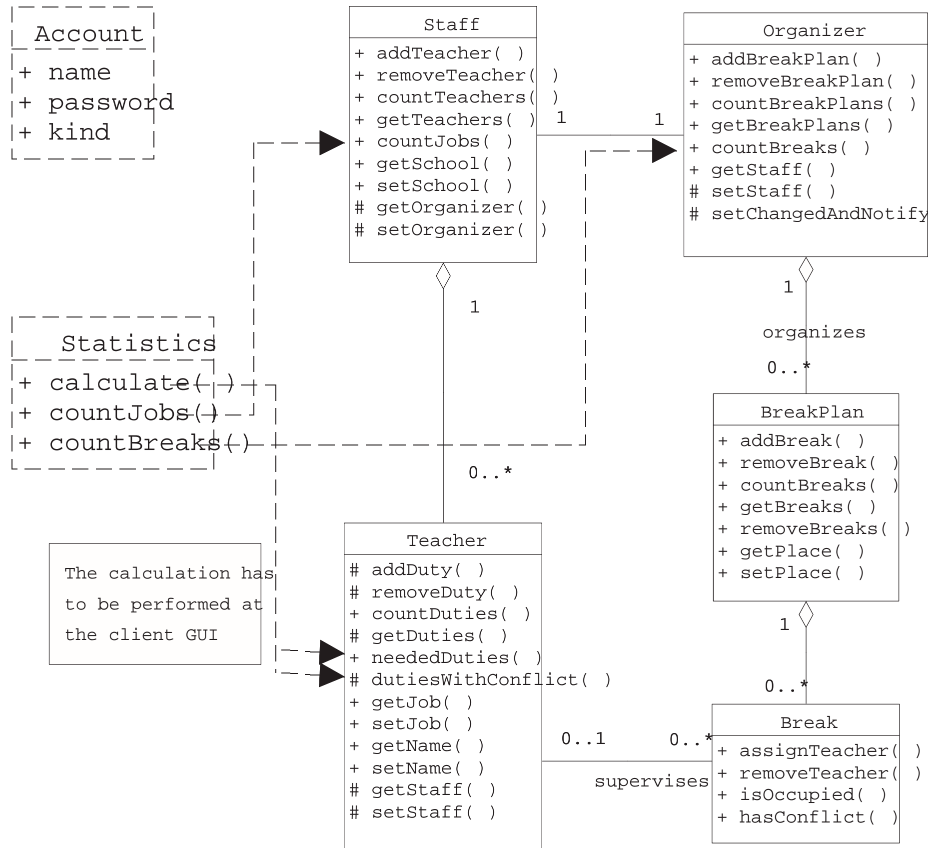

Figure 6: Design Class Diagram of the Break Planner Application

are directly mapped to Java methods.

Instead, we propose to use an *attribute template* that may be associated with the attribute job in an incremental fashion: During analysis, the template only refers to the abstract attribute name. During system design, template parameters specify the way in which way the attribute is to be implemented—in our example this could be done by adding a tag like std-rw to the attribute template which indicates that job is to be implemented by corresponding get() and set() methdods. Once all needed parameters have been provided, the actual translation to low-level design or code may finally be performed.

Analogously, we introduce the concept of an *operation template* and a *relation template*, which obviously have to consider additional information. For better understanding, we list the information needed for the specification of a relation template:

- The *kind* of relation could be, for instance, “aggregation” or “association”.
- The *role names* of the related classes characterize the participants of the relation.
- The *multiplicity* denotes a 1-to-1, 1-to-n, or m-to-n relationship.
- The *navigation properties* indicate which of both classes can navigate over the association. Unidirectional associations are sufficient in most cases.
- The *association management* determines which of both related classes contains operations maintaining the association. If the association is unidirectional, it is reasonable to specify the same class as in navigation properties. Omitting management operations is possible if the connection structure is static or if it is managed by a third party, like a dedicated relation manager, for example.
- The *implementation variant* corresponds to the internal attributes and technical mechanisms used to store and access the association. This may be done by individual reference attributes

in the case of 1-to-1 relations, or by employing the Java Vector class for 1-to-n relations, just to mention a few.

- Additional *qualifiers* may be specified to characterize special properties of the relation, for example, whether the access to the link instances is indexed (in the case of ordered relations).

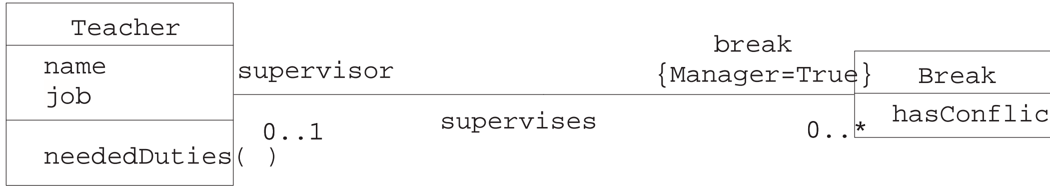

Figure 7: A Template for the Management of Associations

As an example, we consider the m-to-n relation supervises between the classes Teacher and Break. In this case, the class Break is responsible for the management of the relation as can be seen by the parameter {Manager} in Figure 7. Due to this information, additional operations to insert, delete, and get a teacher will be inserted during code generation without the need to specify them explicitly during high-level design.

The proposed *operation templates* were actually very helpful during the implementation of the Break Planner application. On the one hand, the clients of a web-based distributed editor should be offered sufficient functionality by the server to perform effectively and comfortably for their intended purpose. On the other hand, clients should not be allowed to access any additional functionality. This important design guideline is known as the principle of "shallow interfaces" or "loose coupling". It allows to modify operations that are hidden from the clients. In the scope of the Break Planner application, this guideline was applied to offer the classes of a client a reduced functionality as compared to the classes of the server (cf. [BRS98a]).

First, we decided to preceed each operation name with a capital "I" at the beginning. The "I" symbolizes that it is an internal operation which is accessible only by the server. For each of the internal operations which also had to be accessible for clients, we added an additional corresponding external operation (without preceeding "I"). In the actual implementation, the functions visible to the client are only proxies that simply call the corresponding internal functions. As an example, the left side of Figure 8 shows the according interface of the server class Break. Only the operation removeTeacher cannot be called by classes of the clients.

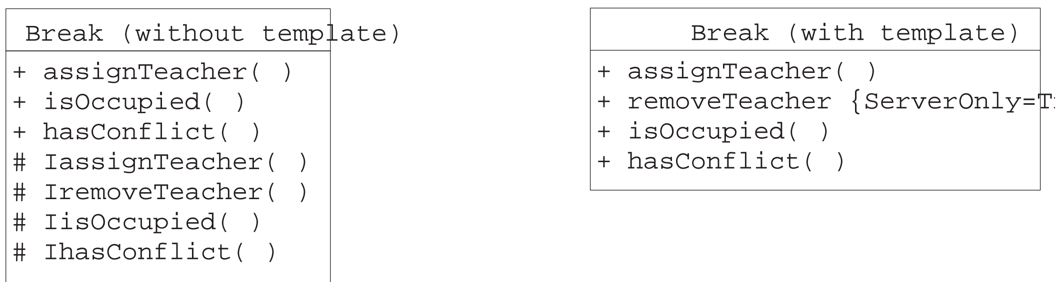

Figure 8: Class Diagram with and without Operation Template

Using operation templates, we may easily enhance the structure and understanding of the class' signature. Server-only operations are tagged by the parameter ServerOnly. The result of this simple application of an operation template is shown on the right hand side of Figure 8.

Note that the use of templates also allows to easily exchange implementation strategies during code generation. A tool, for example, might offer several different strategies to implement a given standard template. This approach is further discussed in the following section.

4 Tool Support

This chapter is reasoning about possible support of the presented concepts and ideas by software development tools. Obviously, the grouping of the features offered by a class as described in Section 3.1 could be easily implemented by well-known tree view techniques. Moreover, the mapping from these structured interfaces to templates and further to the actual implementation is rather straightforward and can thus be automated easily. To its full extent, tools may assist the developer by offering a choice of common templates covering often used concepts like, for instance, aggregation or by allowing to specify and apply custom templates.

Specification A specification tool is needed to support the concepts of feature grouping and templates. A prerequisite is an appropriate description as the one proposed in Section 3. Additionally, the user probably likes to expand and collapse hierarchically structured groups in a comfortable manner.

Documentation The next step is to keep track of the refinement process. Due to its complexity, simple logging mechanisms are not sufficient. Additional information about the involved design and implementation rational needs to be provided by the developer. Thereby, a tool is able to trace which parts of a specification are added within a single refinement step and why this particular step was chosen. It may then use this information for the generation of detailed documentation. Furthermore, a special browser is conceivable that allows developers to trace the refinement process in a comfortable way to reproduce the evolution process in case of changing system requirements. An existing tool dedicated to extract information out of source code for documentation purposes is *JavaDoc* [SUN97a]. We thus propose additional documentation tags to indicate the structure of a class interface as well as the current usage of templates. JavaDoc could use these additional tags to document different abstraction levels of the development information.

Code-Generation For code-generation, numerous additional information is needed. As mentioned above, templates are a predefined refinement step at a level close to the actual implementation and therefore affect code generation. A tool is likely to offer several predefined templates as well as ways to import and export, define or delete custom templates. Another advantage of this fixed translation of specific features to code is the fact that tools focusing on reverse engineering may easily scan for and identify possible templates and thus recognize the corresponding feature. A further aspect of code generation is the way, a tool handles structured grouping of features. For example, the interface may be flattened or separated into different interfaces to realize a multiple implementation relation.

Currently, CASE tools offer only simple functionality to handle increasing complexity. Basically, the developer may display or hide a class' signature while most information regarding the process of code-generation is hidden to the user (as can be seen, for example, in the tool Rational Rose [Rat95]). Although the evolution of a given class diagram from an abstract level during analysis to a very detailed state during implementation is usually supported, the process itself is not documented very well. Thus, the developer cannot easily retrace this process and display the diagram in a former, more abstract state. As a workaround, numerous snapshots need to be taken from a diagram during the different development phases to document all modifications. Instead, as proposed above, a tool should explicitly consider different levels of abstraction by supporting adequate concepts of refinement.

5 Conclusion

In this paper, we introduced a number of simple, yet effective proposals to handle the increasing complexity in class diagrams during the development of object-oriented systems. Hierarchically structured interfaces support different logical views on the features of a class on a given level of abstraction. They are a means to organize these features, thereby improving the understanding of a given interface.

The proposed concepts for refinement, on the other hand, allow to trace the evolution of a class diagram across different levels of abstraction. This allows to document the development process

and the chosen design and implementation rationales. The concept of attribute and implementation templates may be used to abstract from low-level design and implementation details while enhancing the process of code-generation from class diagrams.

However, these concepts need to be supported by future software development tools to realize the potential benefits.

References

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2nd edition, 1994.

[BRS97] Klaus Bergner, Andreas Rausch, and Marc Sihling. Using UML for modeling a distributed Java application. Technical Report TUM-I9735, Technische Universität München, Institut für Informatik, 1997.

[BRS98a] Klaus Bergner, Andreas Rausch, and Marc Sihling. Casting an abstract design into the framework of Java RMI. In *Proceedings of SE:E&P'98*. IEEE Press, 1998.

[BRS98b] Klaus Bergner, Andreas Rausch, and Marc Sihling. A critical look upon UML. In *The Unified Modeling Language*. Springer Verlag, 1998.

[DAC] DACH Group. Universität Hamburg, FB Informatik, AB Softwaretechnik; Johannes-Kepler-Universität, Linz, Austria, Institut für Wirtschaftsinformatik, Doppler-Labor für Software Engineering; GMD Bonn, Schloß Birlinghoven, St. Augustin; UBS Information Technology Laboratory, Zurich, Switzerland.

[Gro97] UML Group. Unified Modeling Language. Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[Jav97] Java security – frequently asked questions `http://java.sun.com/sfaq/index.html`, 1997.

[Obj98] Object International Software. *TogetherJ,* `http://www.oisoft.com`, 1998.

[Rat95] Rational, 3320 Scott Boulevard, Santa Clara, CA 95054 (USA) `http://www.rational.com/`. *Rational Rose*, 1995.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RSLML96] S. Roock, K.-H. Sylla, C. Lilienthal, and A. Müller-Lohmann. Der Pausenplaner – Szenario, CRC-Karten, Systemvision, `http://set.gmd.de/~sylla/dach pap-aufgabe.html`, 1996.

[SUN97a] SUN Microsystems. *javadoc – the Java API Documentation Generator,* `http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javadoc.html`, 1997.

[SUN97b] SUN Microsystems. *RMI – Remote Method Invocation,* `http://java.sun.com/products/jdk/1.1/docs/guide/rmi`, 1997.

[SUN98] SUN Microsystems. *SUN Home Page,* `http://www.sun.com`, 1998.